ISBN 978-1-939133-31-1

Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium

Boston, MA, USA August 10–12, 2022

conference

proceedings

Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium

Boston, MA, USA
August 10–12, 2022

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Association

Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium

August 10–12, 2022
Boston, MA, USA

© 2022 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-939133-31-1

Artifact Evaluation Committee

Artifact Evaluation Committee Co-Chairs

Clémentine Maurice, *CNRS*

Cristiano Giuffrida, *VU Amsterdam*

Artifact Evaluation Committee

Shubham Agarwal, *CISPA Helmholtz Center for Information Security*

Mohsen Ahmadi, *University of California, Los Angeles*

Nikolaos Alexopoulos, *Technische Universität Darmstadt*

Ranya Aloufi, *Imperial College London*

Amit Seal Ami, *College of William & Mary*

Simone Aonzo, *EURECOM*

Daniel Arp, *Technische Universität Braunschweig*

Erin Avllazagaj, *University of Maryland, College Park*

Pierre Ayoub, *EURECOM*

David G. Balash, *The George Washington University*

Shay Berkovich, *BlackBerry*

Jakob Bleier, *TU Wien*

Alejandro Cabrera Aldaya, *Tampere University*

Fabricio Ceschin, *Federal University of Parana*

Weikeng Chen, *University of California, Berkeley*

Hongjun Choi, *Purdue University*

Emilio Coppa, *Sapienza University of Rome*

Pascal Cotret, *ENSTA Bretagne*

Dipanjan Das, *University of California, Santa Barbara*

Giulio De Pasquale, *King's College London*

Luca Degani, *University of Trento and Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche*

Matteo Dell'Amico, *University of Genoa*

Hailun Ding, *Rutgers University*

Dong Du, *Shanghai Jiao Tong University*

Thorsten Eisenhofer, *Ruhr-Universität Bochum*

Alessandro Erba, *CISPA Helmholtz Center for Information Security*

Christof Ferreira Torres, *University of Luxembourg*

Romain Fouquet, *Université de Lille, Inria, CNRS*

Walid Ghandour, *Lebanese University*

Samuel Grayson, *University of Illinois at Urbana–Champaign*

Michele Grisafi, *University of Trento*

Ashish Hooda, *University of Wisconsin—Madison*

Shengtuo Hu, *University of Michigan*

Shan Huang, *Stevens Institute of Technology*

Mazharul Islam, *University of Wisconsin—Madison*

Adriaan Jacobs, *imec-DistriNet, Katholieke Universiteit Leuven*

Jafar Haadi Jafarian, *University of Colorado Denver*

Yilin Ji, *Karlsruhe Institute of Technology*

Kaushal Kafle, *College of William & Mary*

Imtiaz Karim, *Purdue University*

Arslan Khan, *Purdue University*

Hyungsub Kim, *Purdue University*

Soomin Kim, *Korea Advanced Institute of Science and Technology (KAIST)*

Daniel Klischies, *Ruhr-Universität Bochum*

Anne Josiane Kouam, *Inria*

Johannes Krupp, *CISPA Helmholtz Center for Information Security*

Joel Kuepper, *The University of Adelaide*

Georg Land, *Ruhr-Universität Bochum and Deutsches Forschungszentrum für Künstliche Intelligenz*

Ang Li, *Arizona State University*

Song Li, *Johns Hopkins University*

Eleonora Losiouk, *University of Padua*

Keyu Man, *University of California, Riverside*

Damien Marion, *IRISA*

Maryam Masoudian, *Sharif University of Technology, Hong Kong University of Science and Technology (HKUST)*

Guozhu Meng, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences*

Grégoire Menguy, *Université Paris-Saclay, CEA*

Brad Miller, *Google*

Vladislav Mladenov, *Horst Görtz Institute for IT Security*

Eric Mugnier, *University of California, San Diego*

Paul Olivier, *EURECOM*

Kexin Pei, *Columbia University*

Cesar Pereida Garcia, *Tampere University*

Duy-Phuc Pham, *INRIA, IRISA, France*

Goran Piskachev, *Fraunhofer IEM*

Hernan Ponce de Leon, *Bundeswehr University Munich*

Erwin Quiring, *Technische Universität Braunschweig*

Imranur Rahman, *North Carolina State University*

Vidya Lakshmi Rajagopalan, *Stevens Institute of Technology*

Andrew R. Reiter, *Redacted*

Jenni Reuben, *Totalförsvarets forskningsinstitut (FOI)*

Irwin Reyes, *Two Six Technologies*

Michael Rodler, *University of Duisburg-Essen*

Kuheli Sai, *University of Pittsburgh*

Saiful Islam Salim, *University of Arizona*

Solmaz Salimi, *Sharif University of Technology*

Moritz Schloegel, *Ruhr-Universität Bochum*

Vikash Sehwag, *Princeton University*

Omid Setayeshfar, *University of Georgia*

Alex Seto, *Purdue University*

Johnny So, *Stony Brook University*

Marco Squarcina, *TU Wien*

Avinash Sudhodanan, *HUMAN*

Mauro Tempesta, *TU Wien*

Erik Tews, *University of Twente*

Flavio Toffalini, *EPFL*

Guillaume Valadon, *Quarkslab*

Thijs van Ede, *University of Twente*

Alexios Voulimeneas, *imec-DistriNet, Katholieke Universiteit Leuven*

Dawei Wang, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences*

Ding Wang, *Nankai University*

Xiaoguang Wang, *Virginia Tech*

Zhenting Wang, *Rutgers University*

Zhongjie Wang, *Baidu Security*

Noel Warford, *University of Maryland, College Park*

Feng Wei, *University at Buffalo*

Shijia Wei, *The University of Texas at Austin*

Ruoyu Wu, *Purdue University*

Jeffrey Alan Young, *Clemson Univeristy*

Chengyu Zhang, *East China Normal University*

Qingzhao Zhang, *University of Michigan*

Yuchen Zhang, *Stevens Institute of Technology*

Zhiyuan Zhang, *The University of Adelaide*

Yongjun Zhao, *Nanyang Technological University*

Shitong Zhu, *University of California, Riverside*

Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium

August 10–12, 2022
Boston, MA, USA

Wednesday, August 10

Measurement I: Network

Uninvited Guests: Analyzing the Identity and Behavior of Certificate Transparency Bots 1
Brian Kondracki, Johnny So, and Nick Nikiforakis, *Stony Brook University*

Kernel Security

Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability 3
Kyle Zeng, *Arizona State University;* Yueqi Chen, *Pennsylvania State University;* Haehyun Cho, *Arizona State University and Soongsil University;* Xinyu Xing, *Pennsylvania State University;* Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao, *Arizona State University*

In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication 5
Sungbae Yoo, Jinbum Park, Seolheui Kim, and Yeji Kim, *Samsung Research;* Taesoo Kim, *Samsung Research and Georgia Institute of Technology*

Midas: Systematic Kernel TOCTTOU Protection 7
Atri Bhattacharyya, *EPFL;* Uros Tesic, *Nvidia;* Mathias Payer, *EPFL*

Web Security I: Vulnerabilities

Mining Node.js Vulnerabilities via Object Dependence Graph and Query 9
Song Li and Mingqing Kang, *Johns Hopkins University;* Jianwei Hou, *Johns Hopkins University/Renmin University of China;* Yinzhi Cao, *Johns Hopkins University*

FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities 15
Sunnyeo Park and Daejun Kim, *KAIST;* Suman Jana, *Columbia University;* Sooel Son, *KAIST*

Crypto I: Attacking Implementations

TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries 17
Marcel Maehren and Philipp Nieting, *Ruhr University Bochum;* Sven Hebrok, *Paderborn University;* Robert Merget, *Ruhr University Bochum;* Juraj Somorovsky, *Paderborn University;* Jörg Schwenk, *Ruhr University Bochum*

Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design 23
Alon Shakevsky, Eyal Ronen, and Avishai Wool, *Tel-Aviv University*

User Studies I: At-Risk Users

"They Look at Vulnerability and Use That to Abuse You": Participatory Threat Modelling with Migrant Domestic Workers 25
Julia Słupska and Selina Cho, *University of Oxford;* Marissa Begonia, *Voice of Domestic Workers;* Ruba Abu-Salma, *King's College London;* Nayanatara Prakash, *University of Oxford;* Mallika Balakrishnan, *Migrants Organise*

Software Vulnerabilities

How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes 27
Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser, *Technical University of Darmstadt*

Expected Exploitability: Predicting the Development of Functional Vulnerability Exploits 31
Octavian Suciu, *University of Maryland, College Park;* Connor Nelson, Zhuoer Lyu, and Tiffany Bao, *Arizona State University;* Tudor Dumitraș, *University of Maryland, College Park*

Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs 33
Jayakrishna Vadayath, *Arizona State University;* Moritz Eckert, *EURECOM;* Kyle Zeng, *Arizona State University;* Nicolaas Weideman, *University of Southern California;* Gokulkrishna Praveen Menon, *Arizona State University;* Yanick Fratantonio, *Cisco Systems Inc.;* Davide Balzarotti, *EURECOM;* Adam Doupé, Tiffany Bao, and Ruoyu Wang, *Arizona State University;* Christophe Hauser, *University of Southern California;* Yan Shoshitaishvili, *Arizona State University*

Network Security I: Scanning & Censorship

Spoki: Unveiling a New Wave of Scanners through a Reactive Network Telescope 35
Raphael Hiesgen, *HAW Hamburg;* Marcin Nawrocki, *Freie Universität Berlin;* Alistair King, *Kentik;* Alberto Dainotti, *CAIDA, UC San Diego and Georgia Institute of Technology;* Thomas C. Schmidt, *HAW Hamburg;* Matthias Wählisch, *Freie Universität Berlin*

GET /out: Automated Discovery of Application-Layer Censorship Evasion Strategies 37
Michael Harrity, Kevin Bock, Frederick Sell, and Dave Levin, *University of Maryland*

Differential Privacy

Twilight: A Differentially Private Payment Channel Network 41
Maya Dotan, Saar Tochner, Aviv Zohar, and Yossi Gilad, *The Hebrew University of Jerusalem*

Measurement II: Auditing & Best Practices

Building an Open, Robust, and Stable Voting-Based Domain Top List 43
Qinge Xie, *Georgia Institute of Technology;* Shujun Tang, *QI-ANXIN Technology Research Institute;* Xiaofeng Zheng, *QI-ANXIN Technology Research Institute and Tsinghua University;* Qingran Lin, *QI-ANXIN Technology Research Institute;* Baojun Liu, *Tsinghua University;* Haixin Duan, *QI-ANXIN Technology Research Institute and Tsinghua University;* Frank Li, *Georgia Institute of Technology*

Side Channels I: Hardware

Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86 45
Yingchen Wang, *University of Texas at Austin;* Riccardo Paccagnella and Elizabeth Tang He, *University of Illinois Urbana-Champaign;* Hovav Shacham, *University of Texas at Austin;* Christopher W. Fletcher, *University of Illinois Urbana-Champaign;* David Kohlbrenner, *University of Washington*

Web Security II: Fingerprinting

QCSD: A QUIC Client-Side Website-Fingerprinting Defence Framework 47
Jean-Pierre Smith and Luca Dolfi, *ETH Zurich;* Prateek Mittal, *Princeton University;* Adrian Perrig, *ETH Zurich*

Crypto II: Performance Improvements

Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference 51
Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding, *Alibaba Group*

Piranha: A GPU Platform for Secure Computation 55
Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa, *University of California, Berkeley*

OpenSSLNTRU: Faster post-quantum TLS key exchange 57
Daniel J. Bernstein, *University of Illinois at Chicago and Ruhr University Bochum;* Billy Bob Brumley, *Tampere University;* Ming-Shing Chen, *Ruhr University Bochum;* Nicola Tuveri, *Tampere University*

User Studies II: Sharing

Caring about Sharing: User Perceptions of Multiparty Data Sharing 59
Bailey Kacsmar, Kyle Tilbury, Miti Mazmudar, and Florian Kerschbaum, *University of Waterloo*

Hardware Security I: Attacks & Defenses

Jenny: Securing Syscalls for PKU-based Memory Isolation Systems 61
David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard, *Graz University of Technology*

Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks 63
Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering 65
Andrei Tatar, *Vrije Universiteit, Amsterdam;* Daniël Trujillo, *Vrije Universiteit, Amsterdam, and ETH Zurich;* Cristiano Giuffrida and Herbert Bos, *Vrije Universiteit, Amsterdam*

Fuzzing I: Networks

BrakTooth: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing 67
Matheus E. Garbelini, Vaibhav Bedi, and Sudipta Chattopadhyay, *Singapore University of Technology and Design;* Sumei Sun and Ernest Kurniawan, *Institute for Infocomm Research, A*Star*

AmpFuzz: Fuzzing for Amplification DDoS Vulnerabilities 77
Johannes Krupp, *CISPA Helmholtz Center for Information Security;* Ilya Grishchenko, *University of California, Santa Barbara;* Christian Rossow, *CISPA Helmholtz Center for Information Security*

Smart Homes I

SkillDetective: Automated Policy-Violation Detection of Voice Assistant Applications in the Wild 81
Jeffrey Young, Song Liao, and Long Cheng, *Clemson University;* Hongxin Hu, *University at Buffalo;* Huixing Deng, *Clemson University*

Measurement III

A Large-scale Investigation into Geodifferences in Mobile Apps 83
Renuka Kumar, Apurva Virkud, Ram Sundara Raman, Atul Prakash, and Roya Ensafi, *University of Michigan*

Fuzzing II: Low-Level

Morphuzz: Bending (Input) Space to Fuzz Virtual Devices 85
Alexander Bulekov, *Boston University and Red Hat;* Bandan Das and Stefan Hajnoczi, *Red Hat;* Manuel Egele, *Boston University*

Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing 87
Tobias Scharnowski, Nils Bars, and Moritz Schloegel, *Ruhr-Universität Bochum;* Eric Gustafson, *UC Santa Barbara;* Marius Muench, *Vrije Universiteit Amsterdam;* Giovanni Vigna, *UC Santa Barbara and VMware;* Christopher Kruegel, *UC Santa Barbara;* Thorsten Holz and Ali Abbasi, *Ruhr-Universität Bochum*

Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds 91
Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt, *NYU*

Wireless Security

Ghost Peak: Practical Distance Reduction Attacks Against HRP UWB Ranging 93
Patrick Leu and Giovanni Camurati, *ETH Zurich;* Alexander Heinrich, *TU Darmstadt;* Marc Roeschlin and Claudio Anliker, *ETH Zurich;* Matthias Hollick, *TU Darmstadt;* Srdjan Capkun, *ETH Zurich;* Jiska Classen, *TU Darmstadt*

ML I: Federated Learning

Simc: ML Inference Secure Against Malicious Clients at Semi-Honest Cost 97
Nishanth Chandran, Divya Gupta, and Sai Lakshmi Bhavana Obbattu, *Microsoft Research;* Akash Shah, *UCLA*

Thursday, August 11

Deanonymization

Mitigating Membership Inference Attacks by Self-Distillation Through a Novel Ensemble Architecture 101
Xinyu Tang, Saeed Mahloujifar, and Liwei Song, *Princeton University;* Virat Shejwalkar, Milad Nasr, and Amir Houmansadr, *University of Massachusetts Amherst;* Prateek Mittal, *Princeton University*

Synthetic Data – Anonymisation Groundhog Day 103
Theresa Stadler, *EPFL;* Bristena Oprisanu, *UCL;* Carmela Troncoso, *EPFL*

Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses 105
Mojtaba Zaheri, Yossi Oren, and Reza Curtmola, *New Jersey Institute of Technology*

Web Security III: Bots & Authentication

DeepPhish: Understanding User Trust Towards Artificially Generated Profiles in Online Social Networks 107
Jaron Mink, Licheng Luo, and Natã M. Barbosa, *University of Illinois at Urbana-Champaign;* Olivia Figueira, *Santa Clara University;* Yang Wang and Gang Wang, *University of Illinois at Urbana-Champaign*

Crypto III: Private Matching & Lookups

Estimating Incidental Collection in Foreign Intelligence Surveillance: Large-Scale Multiparty Private Set Intersection with Union and Sum 109
Anunay Kulshrestha and Jonathan Mayer, *Princeton University*

Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators 111
Rasoul Akhavan Mahdavi and Florian Kerschbaum, *University of Waterloo*

Incremental Offline/Online PIR 113
Yiping Ma and Ke Zhong, *University of Pennsylvania;* Tal Rabin, *University of Pennsylvania and Algorand Foundation;* Sebastian Angel, *University of Pennsylvania and Microsoft Research*

Passwords

Might I Get Pwned: A Second Generation Compromised Credential Checking Service 115
Bijeeta Pal, *Cornell University;* Mazharul Islam, *University of Wisconsin–Madison;* Marina Sanusi Bohuk, *Cornell University;* Nick Sullivan, Luke Valenta, Tara Whalen, and Christopher Wood, *Cloudflare;* Thomas Ristenpart, *Cornell Tech;* Rahul Chatterjee, *University of Wisconsin–Madison*

Why Users (Don't) Use Password Managers at a Large Educational Institution 119
Peter Mayer, *Karlsruhe Institute of Technology;* Collins W. Munyendo, *The George Washington University;* Michelle L. Mazurek, *University of Maryland, College Park;* Adam J. Aviv, *The George Washington University*

Web Security IV: Defenses

Provably-Safe Multilingual Software Sandboxing using WebAssembly 121
Jay Bosamiya, Wen Shih Lim, and Bryan Parno, *Carnegie Mellon University*

SWAPP: A New Programmable Playground for Web Application Security 125
Phakpoom Chinprutthiwong, Jianwei Huang, and Guofei Gu, *SUCCESS Lab, Texas A&M University*

The Security Lottery: Measuring Client-Side Web Security Inconsistencies 127
Sebastian Roth, *CISPA Helmholtz Center for Information Security;* Stefano Calzavara, *Università Ca' Foscari Venezia;* Moritz Wilhelm, *CISPA Helmholtz Center for Information Security;* Alvise Rabitti, *Università Ca' Foscari Venezia;* Ben Stock, *CISPA Helmholtz Center for Information Security*

ML II

PatchCleanser: Certifiably Robust Defense against Adversarial Patches for Any Image Classifier 129
Chong Xiang, Saeed Mahloujifar, and Prateek Mittal, *Princeton University*

Transferring Adversarial Robustness Through Robust Representation Matching 131
Pratik Vaishnavi, *Stony Brook University;* Kevin Eykholt, *IBM;* Amir Rahmati, *Stony Brook University*

Measurement IV

Measurement by Proxy: On the Accuracy of Online Marketplace Measurements 133
Alejandro Cuevas, *Carnegie Mellon University;* Fieke Miedema, *Delft University of Technology;* Kyle Soska, *University of Illinois Urbana Champaign and Hikari Labs, Inc.;* Nicolas Christin, *Carnegie Mellon University and Hikari Labs, Inc.;* Rolf van Wegberg, *Delft University of Technology*

Hardware Security II: Embedded

RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices 135
Yi He and Zhenhua Zou, *Tsinghua University and BNRist;* Kun Sun, *George Mason University;* Zhuotao Liu and Ke Xu, *Tsinghua University and BNRist;* Qian Wang, *Wuhan University;* Chao Shen, *Xi'an Jiaotong University;* Zhi Wang, *Florida State University;* Qi Li, *Tsinghua University and BNRist*

Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage 137
Yufei Du, *UNC Chapel Hill and University of Rochester;* Zhuojia Shen, Komail Dharsee, and Jie Zhou, *University of Rochester;* Robert J. Walls, *Worcester Polytechnic Institute;* John Criswell, *University of Rochester*

Client-Side Security

Orca: Blocklisting in Sender-Anonymous Messaging 139
Nirvan Tyagi and Julia Len, *Cornell University;* Ian Miers, *University of Maryland;* Thomas Ristenpart, *Cornell Tech*

Adversarial Detection Avoidance Attacks: Evaluating the robustness of perceptual hashing-based client-side scanning 141
Shubham Jain, Ana-Maria Crețu, and Yves-Alexandre de Montjoye, *Imperial College London*

End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage 143
Long Chen, *Institute of Software, Chinese Academy of Sciences;* Ya-Nan Li and Qiang Tang, *The University of Sydney;* Moti Yung, *Google & Columbia University*

Crypto IV: Databases & Logging

Faster Yet Safer: Logging System Via Fixed-Key Blockcipher 147
Viet Tung Hoang, Cong Wu, and Xin Yuan, *Florida State University*

Software Forensics

Back-Propagating System Dependency Impact for Attack Investigation 149
Pengcheng Fang, *Case Western Reserve University;* Peng Gao, *Virginia Tech;* Changlin Liu and Erman Ayday, *Case Western Reserve University;* Kangkook Jee, *University of Texas at Dallas;* Ting Wang, *Penn State University;* Yanfang (Fanny) Ye, *Case Western Reserve University;* Zhuotao Liu, *Tsinghua University;* Xusheng Xiao, *Case Western Reserve University*

Ground Truth for Binary Disassembly is Not Easy 151
Chengbin Pang and Tiantai Zhang, *Nanjing University;* Ruotong Yu, *University of Utah;* Bing Mao, *Nanjing University;* Jun Xu, *University of Utah*

Information Flow

PolyCruise: A Cross-Language Dynamic Information Flow Analysis 153
Wen Li, *Washington State University, Pullman;* Jiang Ming, *University of Texas at Arlington;* Xiapu Luo, *The Hong Kong Polytechnic University;* Haipeng Cai, *Washington State University, Pullman*

SymSan: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis 155
Ju Chen, *UC Riverside;* Wookhyun Han, *KAIST;* Mingjun Yin, Haochen Zeng, and Chengyu Song, *UC Riverside;* Byoungyoung Lee, *Seoul National University;* Heng Yin, *UC Riverside;* Insik Shin, *KAIST*

CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL 157
Flavien Solt, *ETH Zurich;* Ben Gras, *Intel Corporation;* Kaveh Razavi, *ETH Zurich*

FlowMatrix: GPU-Assisted Information-Flow Analysis through Matrix-Based Representation 159
Kaihang Ji, Jun Zeng, Yuancheng Jiang, and Zhenkai Liang, *National University of Singapore;* Zheng Leong Chua, *Independent Researcher;* Prateek Saxena and Abhik Roychoudhury, *National University of Singapore*

Network Security II: Infrastructure

Bedrock: Programmable Network Support for Secure RDMA Systems 163
Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen, *Rice University*

Creating a Secure Underlay for the Internet **165**
Henry Birge-Lee, *Princeton University;* Joel Wanner, *ETH Zürich;* Grace H. Cimaszewski, *Princeton University;* Jonghoon Kwon, *ETH Zürich;* Liang Wang, *Princeton University;* François Wirz, *ETH Zürich;* Prateek Mittal, *Princeton University;* Adrian Perrig, *ETH Zürich;* Yixin Sun, *University of Virginia*

ML III

DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly **169**
Sheng Yu, *University of California Riverside and Deepbits Technology Inc.;* Yu Qu, *University of California Riverside;* Xunchao Hu, *Deepbits Technology Inc.;* Heng Yin, *University of California Riverside and Deepbits Technology Inc.*

Side Channels II

HyperDegrade: From GHz to MHz Effective CPU Frequencies **171**
Alejandro Cabrera Aldaya and Billy Bob Brumley, *Tampere University*

Pacer: Comprehensive Network Side-Channel Mitigation in the Cloud **173**
Aastha Mehta, *University of British Columbia* (*UBC*)*;* Mohamed Alzayat, Roberta De Viti, Björn B. Brandenburg, Peter Druschel, and Deepak Garg, *Max Planck Institute for Software Systems* (*MPI-SWS*)

Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks **175**
Daniel Townley, *Peraton Labs;* Kerem Arıkan, Yu David Liu, and Dmitry Ponomarev, *Binghamton University;* Oğuz Ergin, *TOBB University of Economics and Technology*

Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects **181**
Miles Dai, *MIT;* Riccardo Paccagnella, *University of Illinois at Urbana-Champaign;* Miguel Gomez-Garcia, *MIT;* John McCalpin, *Texas Advanced Computing Center;* Mengjia Yan, *MIT*

Web Security V: Tracking

WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking **183**
Sandra Siby, *EPFL;* Umar Iqbal, *University of Iowa;* Steven Englehardt, *DuckDuckGo;* Zubair Shafiq, *UC Davis;* Carmela Troncoso, *EPFL*

Automating Cookie Consent and GDPR Violation Detection **187**
Dino Bollinger, Karel Kubicek, Carlos Cotrini, and David Basin, *ETH Zurich*

Khaleesi: Breaker of Advertising and Tracking Request Chains **189**
Umar Iqbal, *University of Washington;* Charlie Wolfe, *University of Iowa;* Charles Nguyen, *University of California, Davis;* Steven Englehardt, *DuckDuckGo;* Zubair Shafiq, *University of California, Davis*

Practical Data Access Minimization in Trigger-Action Platforms **191**
Yunang Chen and Mohannad Alhanahnah, *University of Wisconsin–Madison;* Andrei Sabelfeld, *Chalmers University of Technology;* Rahul Chatterjee and Earlence Fernandes, *University of Wisconsin–Madison*

Crypto V: Provers & Shuffling

Polynomial Commitment with a One-to-Many Prover and Applications **193**
Jiaheng Zhang and Tiancheng Xie, *UC Berkeley;* Thang Hoang, *Virginia Tech;* Elaine Shi, *CMU;* Yupeng Zhang, *Texas A&M University*

ppSAT: Towards Two-Party Private SAT Solving **195**
Ning Luo, Samuel Judson, Timos Antonopoulos, and Ruzica Piskac, *Yale University;* Xiao Wang, *Northwestern University*

Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments **197**
Shravan Srinivasan, *University of Maryland;* Alexander Chepurnoy, *Ergo Platform;* Charalampos Papamanthou, *Yale University;* Alin Tomescu, *VMware Research;* Yupeng Zhang, *Texas A&M University*

Friday, August 12

Security Analysis

Loki: Hardening Code Obfuscation Against Automated Attacks 199
Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Julius Basler, *Ruhr-Universität Bochum;* Thorsten Holz, *CISPA Helmholtz Center for Information Security;* Ali Abbasi, *Ruhr-Universität Bochum*

Oops... Code Execution and Content Spoofing: The First Comprehensive Analysis of OpenDocument Signatures 201
Simon Rohlmann, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk, *Ruhr University Bochum*

Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Unity Mobile Games . . . 205
Chaoshun Zuo and Zhiqiang Lin, *The Ohio State University*

SGX I & Side Channels III

Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX 207
Lukas Giner, Andreas Kogler, and Claudio Canella, *Graz University of Technology;* Michael Schwarz, *CISPA Helmholtz Center for Information Security;* Daniel Gruss, *Graz University of Technology*

A Hardware-Software Co-design for Efficient Intra-Enclave Isolation 209
Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen, *Shanghai Jiao Tong University*

SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing 213
Tobias Cloosters, *University of Duisburg-Essen;* Johannes Willbold, *Ruhr-Universität Bochum;* Thorsten Holz, *CISPA Helmholtz Center for Information Security;* Lucas Davi, *University of Duisburg-Essen*

SecSMT: Securing SMT Processors against Contention-Based Covert Channels. 217
Mohammadkazem Taram, *University of California San Diego;* Xida Ren and Ashish Venkat, *University of Virginia;* Dean Tullsen, *University of California San Diego*

Fuzzing III

SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel 219
Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian, *UC Riverside*

Stateful Greybox Fuzzing. 221
Jinsheng Ba, *National University of Singapore;* Marcel Böhme, *Monash University and MPI-SP;* Zahra Mirzamomen, *Monash University;* Abhik Roychoudhury, *National University of Singapore*

Crypto VI

How to Abuse and Fix Authenticated Encryption Without Key Commitment 225
Ange Albertini and Thai Duong, *Google Research;* Shay Gueron, *University of Haifa and Amazon;* Stefan Kölbl, Atul Luykx, and Sophie Schmieg, *Google Research*

Batched Differentially Private Information Retrieval. 227
Kinan Dak Albab, *Brown University;* Rawane Issa and Mayank Varia, *Boston University;* Kalman Graffi, *Honda Research Institute Europe*

One-off Disclosure Control by Heterogeneous Generalization 231
Olga Gkountouna, *University of Liverpool;* Katerina Doka, *National Technical University of Athens;* Mingqiang Xue, *Tower Research;* Jianneng Cao, *Bank Jago;* Panagiotis Karras, *Aarhus University*

User Studies III: Privacy

Security and Privacy Perceptions of Third-Party Application Access for Google Accounts 233
David G. Balash, Xiaoyuan Wu, and Miles Grant, *The George Washington University;* Irwin Reyes, *Two Six Technologies;* Adam J. Aviv, *The George Washington University*

Smart Homes II

SCRAPS: Scalable Collective Remote Attestation for Pub-Sub IoT Networks with Untrusted Proxy Verifier 235
Lukas Petzi, Ala Eddine Ben Yahya, and Alexandra Dmitrienko, *University of Würzburg;* Gene Tsudik, *UC Irvine;* Thomas Prantl and Samuel Kounev, *University of Würzburg*

ML IV: Attacks

AutoDA: Automated Decision-based Iterative Adversarial Attacks 237
Qi-An Fu, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China;* Yinpeng Dong, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China; RealAI;* Hang Su, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China; Peng Cheng Laboratory; Tsinghua University-China Mobile Communications Group Co., Ltd. Joint Institute;* Jun Zhu, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China; RealAI; Peng Cheng Laboratory; Tsinghua University-China Mobile Communications Group Co., Ltd. Joint Institute;* Chao Zhang, *Institute for Network Science and Cyberspace / BNRist, Tsinghua University*

Fuzzing, OS, and Cloud Security

Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies. 239
Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede, *imec-COSIC, KU Leuven*

FixReverter: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing 243
Zenong Zhang and Zach Patterson, *University of Texas at Dallas;* Michael Hicks, *University of Maryland and Amazon;* Shiyi Wei, *University of Texas at Dallas*

Privacy, User Behaviors, and Attacks

PrivGuard: Privacy Regulation Compliance Made Easier 245
Lun Wang, *UC Berkeley;* Usmann Khan, *Georgia Tech;* Joseph Near, *University of Vermont;* Qi Pang, *Zhejiang University;* Jithendaraa Subramanian, *NIT Tiruchirappalli;* Neel Somani, *UC Berkeley;* Peng Gao, *Virginia Tech;* Andrew Low and Dawn Song, *UC Berkeley*

OVRseen: Auditing Network Traffic and Privacy Policies in Oculus VR 247
Rahmadi Trimananda, Hieu Le, Hao Cui, and Janice Tran Ho, *University of California, Irvine;* Anastasia Shuba, *Independent Researcher;* Athina Markopoulou, *University of California, Irvine*

Hardware Security III

Half-Double: Hammering From the Next Row Over 251
Andreas Kogler, *Graz University of Technology;* Jonas Juffinger, *Graz University of Technology and Lamarr Security Research;* Salman Qazi and Yoongu Kim, *Google;* Moritz Lipp, *Amazon Web Services;* Nicolas Boichat, *Google;* Eric Shiu, *Rivos;* Mattias Nissler, *Google;* Daniel Gruss, *Graz University of Technology*

Retbleed: Arbitrary Speculative Code Execution with Return Instructions. 253
Johannes Wikner and Kaveh Razavi, *ETH Zurich*

PISTIS: Trusted Computing Architecture for Low-end Embedded Systems 255
Michele Grisafi, *University of Trento;* Mahmoud Ammar, *Huawei Technologies;* Marco Roveri and Bruno Crispo, *University of Trento*

OS Security & Formalisms

Sapic+: protocol verifiers of the world, unite! 261
Vincent Cheval, *Inria Paris;* Charlie Jacomme, *CISPA Helmholtz Center for Information Security;* Steve Kremer, *Université de Lorraine LORIA & Inria Nancy;* Robert Künnemann, *CISPA Helmholtz Center for Information Security*

ML V: Principles & Best Practices

On the Security Risks of AutoML. 263
Ren Pang and Zhaohan Xi, *Pennsylvania State University;* Shouling Ji, *Zhejiang University;* Xiapu Luo, *Hong Kong Polytechnic University;* Ting Wang, *Pennsylvania State University*

User Studies IV: Policies & Best Practices

Where to Recruit for Security Development Studies: Comparing Six Software Developer Samples. 265
Harjot Kaur, *Leibniz University Hannover;* Sabrina Klivan, *CISPA Helmholtz Center for Information Security;* Daniel Votipka, *Tufts University;* Yasemin Acar, *Max Planck Institute for Security and Privacy and George Washington University;* Sascha Fahl, *CISPA Helmholtz Center for Information Security and Leibniz University Hannover*

SGX II

MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties **267**
Guoxing Chen, *Shanghai Jiao Tong University;* Yinqian Zhang, *Southern University of Science and Technology*

ELASTICLAVE: An Efficient Memory Model for Enclaves **269**
Jason Zhijingcheng Yu, *National University of Singapore;* Shweta Shinde, *ETH Zurich;* Trevor E. Carlson and Prateek Saxena, *National University of Singapore*

Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks **271**
Andreas Kogler and Daniel Gruss, *Graz University of Technology;* Michael Schwarz, *CISPA Helmholtz Center for Information Security*

Network Security III: DDoS

Anycast Agility: Network Playbooks to Fight DDoS **273**
A S M Rizvi, *USC/ISI;* Leandro Bertholdo, *University of Twente;* João Ceron, *SIDN Labs;* John Heidemann, *USC/ISI*

Regulator: Dynamic Analysis to Detect ReDoS **281**
Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna, *University of California, Santa Barbara*

Zero Knowledge

Aardvark: An Asynchronous Authenticated Dictionary with Applications to Account-based Cryptocurrencies . . . **283**
Derek Leung, *MIT CSAIL;* Yossi Gilad, *Hebrew University of Jerusalem;* Sergey Gorbunov, *University of Waterloo;* Leonid Reyzin, *Boston University;* Nickolai Zeldovich, *MIT CSAIL*

Zero-Knowledge Middleboxes **287**
Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish, *NYU*

Efficient Representation of Numerical Optimization Problems for SNARKs **291**
Sebastian Angel, *University of Pennsylvania and Microsoft Research;* Andrew J. Blumberg, *Columbia University;* Eleftherios Ioannidis and Jess Woods, *University of Pennsylvania*

Experimenting with Collaborative zk-SNARKs: Zero-Knowledge Proofs for Distributed Secrets **295**
Alex Ozdemir and Dan Boneh, *Stanford University*

Software Security

Detecting Logical Bugs of DBMS with Coverage-based Guidance **297**
Yu Liang, *Pennsylvania State University;* Song Liu, *Pennsylvania State University and Qi-AnXin Tech. Research Institute;* Hong Hu, *Pennsylvania State University*

Debloating Address Sanitizer **301**
Yuchen Zhang, *Stevens Institute of Technology;* Chengbin Pang, *Nanjing University;* Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu, *Stevens Institute of Technology*

Side Channels IV

Automated Side Channel Analysis of Media Software with Manifold Learning **305**
Yuanyuan Yuan, Qi Pang, and Shuai Wang, *The Hong Kong University of Science and Technology*

ML VI: Inference

Membership Inference Attacks and Defenses in Neural Network Pruning **307**
Xiaoyong Yuan and Lan Zhang, *Michigan Technological Unviersity*

Are Your Sensitive Attributes Private? Novel Model Inversion Attribute Inference Attacks on Classification Models **309**
Shagufta Mehnaz, *The Pennsylvania State University;* Sayanton V. Dibbo and Ehsanul Kabir, *Dartmouth College;* Ninghui Li and Elisa Bertino, *Purdue University*



A Artifact Appendix

A.1 Abstract

In our paper, "Uninvited Guests: Analyzing the Identity and Behavior of Certificate Transparency Bots", we curated an extensive dataset of web requests originating from bots monitoring Certificate Transparency (CT) logs. In total, we recorded over 1.5 million requests from CT bots, originating from 31,898 unique IP addresses. To assist in the understanding and further exploration of this previously-unexplored population of bots, we are releasing our dataset and domain generation script to researchers.

We observed that CT bots can be subdivided into distinct groups based on the types of hosts they target, each with varying behaviors. Using our provided dataset, one can analyze these subsets of CT bots, including the populations of each group and characteristics of the web requests they transmit.

A.2 Artifact check-list (meta-information)

- **Publicly available:**
  - Dataset: https://zenodo.org/record/6677235#.YrH-o3jMJes
  - Domain generation script: https://zenodo.org/record/6818616#.YsxXLy-B0iY
- **Data licenses:** Licenced under *Creative Commons Attribution 4.0 International*
- **DOI:**
  - Dataset: 10.5281/zenodo.6677235
  - Domain generation script: 10.5281/zenodo.6818616

A.3 Description

A.3.1 How to access

Download our dataset and domain generation script using the Zenodo links in Section A.2.

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

- Python3
- Python Pandas library

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

N/A

A.5 Evaluation and expected results

In our paper, we studied the behavior and identity of Certificate Transparency (CT) bots, curating a dataset consisting of over 1.5 million web requests from bots consuming CT logs. We found that this previously-unexplored population of web bots can be sub-divided into groups that target specific types of hosts based on the content of their domains names–with each subset exhibiting unique behaviors.

In addition to providing our full dataset of CT bot web requests, we have also included an example analysis script that can be used to reproduce a number of general statistics and a table listed in our paper. To do this, download our dataset and analysis script from the Zenodo repository listed in Section A.2. Next, using a Python3 interpreter, install the Python Pandas library (listed in the included *requirements.txt* file). Finally, run the example analysis script and review the outputted results on the terminal, which include the population sizes of each CT bot subset as well as a table listing the most common file paths requested by bots in each group.

To assist the research community in reproducing our CTPOT system, we are also releasing a Python script that can be used to generate pseudo-random subdomains to be advertised on CT through the generation of TLS certificates. After downloading the script from the Zenodo link listed in Section A.2, simply execute it using a Python3 interpreter. A unique pseudo-random subdomain will then be printed onto the terminal.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact requires machines with x86_64 architecture. At least 2 logical cores and 2 GB RAM is required for running the experiments. Since the experiment is resource-consuming, more cores and RAM settings are recommended. The artifact has been containerized, so it runs on most Linux-based operating systems. It has been verified to work on Ubuntu-20.04.

Our paper is about empirically evaluating the reliability enhancement brought by kernel exploit stabilization techniques; the empirical experiment forms the foundation for our paper.

To validate the experiment, one can repeat the experiment included in the artifact and compare the result with what we present in the paper. Since our experiment result can be slightly affected by the underlying hardware, we expect the result on another machine to be slightly different from what is in the paper. However, the effect of each exploit stabilization technique should not change. In other words, if a technique improves exploit reliability for a specific CVE in the paper, it should behave the same in repeated experiments. However, the improvement may be slightly different.

A.2 Artifact check-list (meta-information)

- **Binary:** Compiled vulnerable Linux kernels are included. New vulnerable kernels can be compiled as well using scripts/kernel_builder/build_kernels.py.
- **Data set:** The artifact requires a dataset of vulnerable Linux kernels and corresponding exploits. They are included in exploit_env/CVEs/.
- **Run-time environment:** The artifact depends on "docker" software. It requires a Linux-based host OS to build the container image. It has been verified to work on Ubuntu-20.04. The OS inside the container is Ubuntu-18.04. root access on the host OS is required.
- **Execution:** The experiment should be run on a machine without other processes running. The existence of other processes may interfere with the experiment and affect the result.
- **Metrics:** The metric used in the experiment is the success rate of exploits.
- **Output:** The output of the experiment is the success rate of each exploit. The number of success/failure runs is saved in a JSON file in the output folder.
- **Experiments:** To prepare and run the experiment, one needs to 1. clone the artifact repository from https://github.com/sefcom/KHeaps, 2. build the docker image as instructed in README.md, and 3. run an evaluation experiment for each CVE as instructed.

  The expected result is included in the paper. We expect slightly different success rates for each exploit. However, the effect of each exploit stabilization technique should be the same. In other words, if a technique improves reliability in the paper, the behavior should stay the same in repeated experiments with a slightly different improvement. The same applies to the cases where techniques hurt exploit reliability.
- **How much disk space required (approximately)?:** We expect the whole experiment to take about 20GB disk space after disabling logging (the "-nl" option in "vuln_tester.py").
- **How much time is needed to prepare workflow (approximately)?:** We containerized the whole experiment. It takes about 10-15 minutes to build a disk image for the VM and docker image for the evaluation.
- **How much time is needed to complete experiments (approximately)?:** To complete the 2CPU+2GB RAM experiment (each VM configured with 2 virtual CPU and 2GB RAM), it requires 1680 CPU days. The time needed can be reduced by increasing the number of CPUs. For example, it can be finished in 42 days with a 40-core machine.
- **Publicly available (explicitly provide evolving version reference)?:** The artifact is publicly available at https://github.com/sefcom/KHeaps
- **Code licenses (if publicly available)?:** MIT license.
- **Data licenses (if publicly available)?:** MIT license.
- **Archived (explicitly provide DOI or stable reference)?:** Stable reference on GitHub: https://github.com/sefcom/KHeaps/tree/22b35da5f9f259f5cc8f349da9f791d9428295e4.

A.3 Description

A.3.1 How to access

Clone git repository from https://github.com/sefcom/KHeaps/tree/22b35da5f9f259f5cc8f349da9f791d9428295e4.

A.3.2 Hardware dependencies

N/A.

A.3.3 Software dependencies

The experiment requires a Linux-based OS to build. Ubuntu-20.04 is preferred.

One of the experiments requires nested-kvm parameter in kvm-intel kernel module. One can check whether it is enabled by checking /sys/module/kvm_intel/parameters/nested. If it is enabled, the pseudo file should return "Y".

The experiment depends on "docker" software.

A.3.4 Data sets

The dataset is included in the public KHeaps repository. It consists of two parts: 1. vulnerable kernels are pre-compiled and included in the repository. 2. kernel exploits are included in "poc" folders.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Use the following command to build the docker image.

1. git clone https://github.com/sefcom/KHeaps
2. cd KHeaps
3. cd scripts/create-image/ && ./create-image.sh && cd ../..
4. docker build -t kheap .

The above process takes about 10 minutes to finish.

At this stage, a docker image called "kheap" should be created. One can verify this by making sure its existence in the output of "docker images".

A.5 Experiment workflow

The experiment aims to evaluate the success rates of exploits against vulnerable kernels. For each CVE, it compiles all the corresponding exploits first and then launches VMs with the vulnerable kernel. It then copies exploits into the VMs using ssh and runs exploits inside the VMs until the VMs crash. The VM monitor will extract the crash logs and determine whether the exploits succeed or not. We regard an exploit as successful if the VM crashes at an attacker-controlled program counter, which demonstrates the control flow hijacking capability of the exploit.

A.6 Evaluation and expected results

Main claim: Exploits equipped with the combo technique outperforms realworld exploits in terms of reliability. This can be verified by running realworld exploits and combo exploits and comparing their success rates. In our evaluation, the success rates of realworld and combo exploits are 54.30% and 91.15% (67.86% improvement). In repeated experiments, we expect combo exploits to have at least 50% improvement over realworld exploits.

Key results:

- Defragmentation improves reliability for OOB exploits. We expect exploits equipped with Defragmentation technique to have a significantly higher success rate compared with baseline exploits. This can be verified by running baseline exploits and exploits equipped with Defragmentation technique.
- Defragmentation may hurt reliability for UAF or DF exploits. We expect that Defragmentation does not significantly improve reliability for UAF and DF exploits and significantly hurts the reliability for some of them. For example, CVE-2017-2636.
- Heavy workload hurts exploit reliability, but exploits can still achieve high success rates. This can be verified by running exploits in both idle and busy settings. One should observe exploit success rate degradation in busy settings and that more than half exploits equipped with Multi-Process Heap Spray can achieve more than 90% success rates.
- Multi-Process Heap Spray generally outperforms Single-Thread Heap Spray. This can be verified by running both Multi-Process Heap Spray and Single-Thread Heap Spray exploits. We expect Multi-Process Heap Spray to outperform Single-Thread Heap Spray in all settings with one exception: CVE-2017-6074 in idle settings, potentially also in busy settings.

A.7 Experiment customization

To evaluate exploits for a new CVE, one needs to add a new folder in "CVEs" folder and specify its maximum runtime in "setup.json".

To limit the evaluation to some specific exploits, one can add filters in "make_pocs" function in "vuln_tester.py" script.

A.8 Notes

N/A.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

A Artifact Appendix

A.1 Abstract

Our artifact provides code and binaries and scripts to reproduce experimental results in the paper. As for benchmarks, we demonstrated experimental results on two real machines (Mac mini based on M1 chip, and rpi3), but our artifact has been prepared to run only on Mac mini, since rpi3 lacks a support of ARM Pointer Authentication (PA). Other results than benchmarks can be reproducible on host PC machines.

A.2 Artifact check-list (meta-information)

- **Algorithm:** compiler instrumentation, pointer analysis
- **Program:** Linux kernel, LLVM plugin, GCC plugin, python scripts (all sources and binaries included)
- **Compilation:** LLVM 9.0, Modified GCC 7.3 (binaries and sources included)
- **Transformations:** Security check insertion implemented as a GCC plugin.
- **Binary:** Pre-built root file system, kernel images with various configurations.
- **Run-time environment:** Ubuntu 18.04 or 20.04. Host environment, not virtual environment, is recommended.
- **Hardware:** Mac mini with M1 chip
- **Security, privacy, and ethical concerns:** Some of shared codes are subject to intellectual property. Please do not redistribute them.
- **Output:** static analysis results of context analyzer and static validator, benchmark results on Mac mini.
- **How much disk space required (approximately)?:** The artifact repository takes up around 5GB.
- **How much time is needed to prepare workflow (approximately)?:** It takes about 1-2 hours to prepare.
- **How much time is needed to complete experiments (approximately)?:** It takes about 1 hour to complete.
- **Publicly available (explicitly provide evolving version reference)?:** Some codes, which have no issue of intellectual property, will be available at https://github.com/SamsungLabs/PALinux/ soon.

A.3 Description

A.3.1 How to access

You can access all materials for the artifact evaluation through a repository in the Bitbucket: https://bitbucket.org/jinb-park/pal-ae/. Note that this is a private repo because we cannot open the source code to the public yet due to an issue of intellectual property. Reviewers can access this repo by using the SSH key posted on "Artifact access" section in the hotcrp submission site.

A.3.2 Hardware dependencies

It is required to have a physical access to a mac mini built on M1 chip for reproducing benchmarks. Also, it requires a USB-to-C cable and a HDMI cable for connection between the mac mini and host PC.

A.3.3 Software dependencies

We have confirmed this artifact on Ubuntu 18.04/20.04 host machines, not virtual guest machines. We also tried our artifact on virtual guests but found that some of experiments can go wrong. So we recommend running this on host machines if possible.

A.4 Installation

First of all, copy the SSH key content into a file (e.g., pal_rsa), and then type the following command:

```
$ chmod 600 pal_rsa
$ GIT_SSH_COMMAND='ssh -i pal_rsa -o
    IdentitiesOnly=yes' git clone git@bitbucket.
    org:jinb-park/pal-ae.git
```

Next, follow the guide, README.md, in the repository.

A.5 Experiment workflow

Our experiment workflow is twofold.
First, for functional evaluation, run the context analyzer to get a CFI precision report and a guide for dynamic contexts; then build linux kernel along with the guide; lastly, run the static validator on the built kernel binary to find out insecure uses of PA (Pointer Authentication) instructions. See "full-workflow/README.md" to get to know all instructions needed for it.
Second, for reproducing key results, we put appropriate prebuilt files as well as a README file that contains required instructions in each directory in the repository. (analyzer/, precision/, static-validator/, benchmarks/)

A.6 Evaluation and expected results

Key results that are reproducible are:

- **Context analyzer (Table 6):** We present the prebuilt llvm bitcode file for a whole kernel binary and the source code of context analyzer in the form of LLVM plugin. See "analyzer/README.md" for detail.
- **CFI precision (Table 2 and Table 3):** It shows how CFI precision gets better as dynamic contexts are used, and reproduces Table 2 and Table 3 in our paper. See "precision/README.md" for detail.
- **Static validator (Section 4.5 - Results.):** We give an in-depth analysis for violations that our validator found (Section 4.5 Results). Also, we present two prebuilt kernel binaries (iOS and PAL) and validator's code written in python, allowing to run our validator. See "static-validator/README.md" for detail.

- **Benchmarks on Mac mini (Table C1 and Section 6.3 - Performance Overhead)):** We present kernel images built with or without PAL, and a root filesystem that contains macro- and micro-benchmarks, and a helper script to run them on Mac mini. See "benchmarks/README.md" for detail.

A.7 Experiment customization

Each README.md in the bitbucket repo explains on how to customize experiments in detail.

A.8 Notes

The results of benchmarks can fluctuate. Even when we used the same mac mini, we saw that its results can vary around 2 times slower or faster at maximum, especially for benchmarks that take a relatively short time.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

For Midas, we present an artifact including the source code and binaries for the prototype based on Linux, an exploit which demonstrate that Midas mitigates a real CVE, and benchmarks for evaluating Midas' performance, and scripts which simplify the process. In the following sections, we describe the artifact, its requirements and how to run it, and what the expected results are. Visit the project website https://hexhive.epfl.ch/midas for more details.

A.1 Description

The primary artifact for this paper is the code implementing Midas on the Linux kernel (v5.11), available on GitHub. We also provide a disk image suitable for recreating experiments from this paper, containing the kernel as both source code and as compiled binaries. The disk image contains the CVE exploit used to test correctness in the paper, all benchmarks evaluated in the paper, and scripts to run these. This image allows recreation of all emperical evidence presented in the paper's evaluation. Finally, we provide further information on the project website including a detailed description of the artifact, its contents, how to run it and expected outputs.

- Source code: https://github.com/HexHive/midas
- Disk image: https://zenodo.org/record/5753026
- Project website: https://hexhive.epfl.ch/midas

A.1.1 Hardware Dependencies

You can run the disk image within a QEMU virtual machine to test functionality. The host machine requies around 100GiB free disk space and at least 8GiB memory. You should run the disk image on a real machine for performance tests. Our Midas prototype supports machines with 64-bit x86 processors, and the results in the paper were obtained on a machine with an Intel i7-9700 CPU. Further, the real machine requires an empty 1TiB disk, and a EUFI-enabled motherboard. In both setups, a SSD is preferred for storage, as it leads to faster compilation should you choose to re-compile the kernel. Evaluating the Nginx benchmark requires a second, networked machine to act as a load generator.

A.1.2 Software Dependencies

Running the Midas disk image requires a guest operating system which supports running QEMU. The image was tested on QEMU version `4.2.1` on a machine running Ubuntu 20.04 with Linux kernel version `5.4.0-88-generic`. Other virtualization software should also be supported, but the instructions focus on QEMU. Running the disk image on real hardware requires no special software support, apart from a tool to write the image to a disk. On Linux, we can use `dd`.

A.2 Installation

The installation procedure includes downloading and uncompressing the provided compressed disk image, then either running a VM directly from this image, or by writing the image to a disk and booting from it.

On Linux, the following command extracts the image.

```
pv ae.img.xz | unxz -T <num threads> > ae.img
```

The uncompressed disk image can then either be run with QEMU, or written to a real disk. To run with QEMU, an example command is shown below.

```
qemu-system-x86_64                    \
  -m 4G                               \
  -cpu host                           \
  -machine type=q35,accel=kvm         \
  -smp 4                              \
  -drive format=raw,file=ae.img       \
  -display default                    \
  -vga virtio                         \
  -show-cursor                        \
  -bios /usr/share/ovmf/OVMF.fd       \
  -net user,hostfwd=tcp::2222-:22     \
  -net nic
```

To run on real hardware, copy the image to a real disk using the command shown below, then install into the machine and start it.

```
dd if=ae.img of=/dev/<disk> bs=100M
```

A.3 Experiment Workflow

The experimental workflow compares the modified Midas kernel with the baseline Linux kernel. Detailed steps are available on the website at https://hexhive.epfl.ch/midas/docs/ae.html. You can validate the artifact by executing the following steps:

- Check that the code modifications described in the paper correspond to the code.
- Compile the code to re-create the kernel binary.
- Run a script to check that a CVE exploit is mitigated, as claimed in the paper.
- Run scripts to execute the benchmarks presented in the paper, to verify their reported performance.

For the CVE exploitation test, the dmesg output must be checked to ensure that Midas prevents exploitation. For the performance experiments, the results must be compiled and compared to get the Midas' relative performance. The general workflow is:

- boot with the correct kernel (baseline or Midas),
- run the script for the benchmark/CVE exploit,
- reboot with the other kernel, and
- run the same script again.

A.4 Expected Results

Midas is evaluated to demonstrate effective mitigation of double-fetch bugs with low overhead. The artifact enables you to verify this claim, that the prototype provides the claimed protection and that it performs as claimed. We demonstrate the first property by including checks in the kernel and running an exploit for CVE-2016-6516 to demonstrate its mitigation. The remaining benchmarks measure performance, either as operations per second or as time taken to finish each operation. Below, we describe how to interpret the outputs of running the exploit and benchmarks.

Midas protects the kernel against double-fetch bugs, and in particular mitigates an exploit for CVE-2016-6516. In our prototype, you will execute the exploit with and without Midas' protections. When run with the baseline kernel, the exploit is triggered, and the string `"Triggered bug: CVE-2016-6516!"` will be printed to `dmesg` output. With the Midas kernel, the string is never printed.

We also run kernel-intensive benchmarks which demonstrate that Midas has a low runtime overhead. Our artifact also contains the performance benchmarks used for testing Midas' performance. The benchmarks must be run separately with both the baseline and Midas kernel. We include a script to plot the relative performance vs. the baseline kernel. Midas' performance is strongly dependent on the CPU used for evaluation, and exact performance values can vary significantly. However, we expect the trends of performance across benchmarks to roughly follow the following limits.

- Microbenchmarks see results in line with paper.
- NPB benchmarks experience 0-5% overhead, and should follow the numbers from the paper.
- PTS benchmarks - openssl, git, pybench, redis see an overhead <1%.
- PTS benchmarks - apache sees a overhead < 10-15%.
- PTS benchmarks - IPC benchmark sees overhead < 5%.
- Nginx shows a constant overhead as request size changes, until the network link is saturated.

The setup for breaking down Midas' overhead is complicated, and omitted from this artifact.

A.5 Artifact meta-information

- **Program:** NASA Parallel Benchmarks (NPB), Phoronix Test Suite (PTS), Nginx, the Linux kernel, and exploits for CVE-2016-6516. All benchmarks and code are publicly available, and are installed in the provided disk image.
- **Binaries:** The disk image provides the compiled Linux kernel (v5.11) with and without Midas' protections.
- **Hardware:** For functionality evaluation, one machine with 100GiB free disk space, and QEMU (version 4.2). For results reproduction, one machine with modern Intel x86 CPU, and a free 1TiB disk. In both setups, a SSD is preferred.
- **Run-time state:** The disk image includes a program for fixing CPU frequency, eliminating run-time variance. This only works on native hardware, not QEMU.
- **Metrics:** NPB workloads report execution rate. PTS workloads report either execution time or operation rate. Nginx reports both request rate and throughput.
- **Output:** Most benchmarks and tests output to a console.
- **Experiments:** Experiments have been prepared within the disk image, and can be run using provided scripts.
- **How much time is needed to prepare workflow (approximately)?:** 3-4 hours, on a machine with an SSD.
- **How much time is needed to complete experiments (approximately)?:** For performance evaluation, approx. 8 hours.
- **Publicly available?:** All code is publicly available.
- **Code license:** GPL v2.0
- **Archived (provide DOI or stable reference)?:** DOI 10.5281/zenodo.5753026 available at https://zenodo.org/record/5753026.



A Artifact Appendix

A.1 Abstract

In the paper, we propose flow- and context-sensitive static analysis with hybrid branch-sensitivity and points-to information to generate a novel graph structure, called Object Dependence Graph (ODG), using abstract interpretation. ODG represents JavaScript objects as nodes and their relations with Abstract Syntax Tree (AST) as edges, and accepts graph queries—especially on object lookups and definitions—for detecting Node.js vulnerabilities.

We implemented an open-source prototype system, called ODGEN, to generate ODG for Node.js programs via abstract interpretation and detect vulnerabilities. Our evaluation of recent Node.js vulnerabilities shows that ODG together with AST and Control Flow Graph (CFG) is capable of modeling 13 out of 16 vulnerability types. We applied ODGEN to detect six types of vulnerabilities using graph queries: ODGEN correctly reported 180 zero-day vulnerabilities, among which we have received 70 Common Vulnerabilities and Exposures (CVE) identifiers so far.

In this artifact evaluation, we claim that OPGEN is capable of detecting all six types of vulnerabilities and found all the zero-day vulnerabilities.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Mining Node.js Vulnerabilities via Object Dependence Graph and Query
- **Data set:** We use the self-generated dataset and it is included in the docker image
- **Run-time environment:** Ubuntu 20.04 is recommanded and tested. The main software dependencies are Python 3.7+, pip, npm, and Node.js 12+
- **Run-time state:** No
- **Metrics:** Number of detected vulnerable packages
- **Output:** The testing results are located in the "logs" folder of the running directory. All the detected vulnerable packages will be output to the "succ.log" file; All the un-detected packages will be output to the "results.log" file. You can get the number of the successfully detected packages by running "cat ./logs/succ.log | wc -l", during or after the running process.
- **Experiments:** You can download and load the docker, or set up the environment from the source code. Then run the pre-written scripts and see the results.
- **How much disk space required (approximately)?:** 10GB
- **How much time is needed to prepare workflow (approximately)?:** 10 to 30 mins
- **How much time is needed to complete experiments (approximately)?:** 200 mins
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GPL v3.0
- **Data licenses (if publicly available)?:** GPL v3.0
- **Archived (provide DOI)?:**

A.3 Description

A.3.1 How to access

We provide two methods for testing, loading the docker image is highly recommanded:

- A docker image

  We uploaded our docker to Docker Hub. You can pull it by running

  docker pull iamthesong/odgen:latest

  Then you can attach to this docker by running

  docker run -it iamthesong/odgen bash

  After loading it, you should be able to see the environment

- A repository for the source code

  If you are not able to access the virtual machine and can not load the docker image, you can also try to clone our source code from the GitHub repository https://github.com/Song-Li/ODGen/tree/24d68fa810cae8c028cf36f269461e178c198c98 (commit hash: 24d68fa810cae8c028cf36f269461e178c198c98) and follow the instructions in the README.md to set up the environment.

A.3.2 Hardware dependencies

Recommended

- CPU: 16 cores
- Memory: 16GB

Minimum

- CPU: 4 cores
- Memory: 4GB

A.3.3 Software dependencies

If you want to start with the source code, Ubuntu 20.04 is recommended. This artifact requires Python 3.7+, pip, npm, and Node.js 12+.

A.4 Installation

A.4.1 Docker image

We prepared a docker image on Docker Hub. You can follow the commands mentioned in section A.3.1 to download the load the docker.

A.4.2 Source code

Setup the Environment If you want to start with the source code, we recommend you to use Ubuntu 20.04., you can simply cd into the source code folder and install the software dependencies by running:

./ubuntu_setup.sh

After the packages are successfully installed, you can setup the environment by running:

./install.sh

The script **install.sh** will install a list of required Python and Nodejs dependencies. Once finished, the environment is setup and we are ready to go.

Verify the Installation You can run the script **odgen_test.py** to verify the installation. The command is:

python3 ./odgen_test.py

If the environment is successfully set up, you should be able to see the tests are finished without errors. The end of the outputs should be like:

Ran 3 tests in XXXs

OK

A.5 Experiment workflow

As we claimed in the Abstract section and the Contributions part of section 1 in our paper, our main claim that needed to be evaluated is we found 43 application-level and 137 package-level zero-day vulnerabilities. Our tool can successfully found the vulnerabilities of those packages. We prepared the dataset and the related scripts to run our tool on top of the packages.

Besides the main claim, other evaluation results, including the performance, the code coverage, and the false-negative rate of our paper are also reproducible and reproduced by the reviewers. I will also include the steps and datasets to reproduce the related evaluation results in the next section.

A.5.1 File organization of our Docker Image

Once you log into the docker, all the files and folders are organized as follows:

```
.
|--projs: the source code and libs of our tool.
|--packages: all the zero-day vulnerable packages detected by our tool.
  |--code_exec: packages with zero-day arbitrary code execution vulnerabilities
    |--XX: package-name@version
      |--cve.txt: if it exists, it indicates the CVE identifier
      |--run.sh: a script to detect the zero-day vulnerability
  |--ipt: packages with zero-day internal property tampering vulnerabilities
  |--os_command: packages with zero-day OS command injection vulnerabilities
  |--path_traversal: packages with zero-day path traversal vulnerabilities
  |--proto_pollution: packages with zero-day prototype pollution vulnerabilities
  |--xss: packages with zero-day XSS vulnerabilities
|--examples: a few simple vulnerable examples
  |--pp_example.js: the prototype pollution example
  |--run_proto_pollution.sh: detect prototype pollution of pp_example.js
  |--motivating_example.js: the motivating example mentioned in the paper
  |--run_ipt.sh: detect internal property tampering of motivating_example.js
  |--run_os_command.sh: detect taint-style vulnerability of motivating_example.js
  |--clean.sh: clean up log files
|--back_up: recovery files (do not touch)
```

A.5.2 Dataset

Dataset 1: Zero-day vulnerable packages

- **dataset:** The 174 zero-day vulnerable packages that found by our tool. (Note that after our reporting, there are eight packages that are unpublished from NPM. Currently, we only have source code for 173 packages + one package, which is unpublished but cached on our server.)
- **location:** ∼/packages
- the CVEs they got: ∼/packages/xx/package-name@version/cve.txt (if exists)
- a script that runs the analysis on each of these folders/projects and detects the vulnerabilities: ∼/packages/xx/package-name@version/run.sh

  where xx = code_exec, ipt, os_command, path_trasversal, proto_pollution, and xss.

Note that considering the large size of the dataset, we are not able to upload the dataset to the GitHub repository. We uploaded the zipped dataset to Google Drive and if you are testing it by the source code, please download it, unzip it, and put it in the root directory of your machine.

Dataset 2: Legacy vulnerable packages

- **dataset:**The legacy vulnerable packages dataset mentioned in Section 6.3 of the paper, including 75 command injection vulnerable packages, 31 code execution vulnerable packages, 52 prototype pollution vulnerable packages, 87 path traversal vulnerable packages, and 11 internal property tampering (IPT) vulnerable packages.
- **location:** We uploaded it as a zip file to the GitHub Repo (https://github.com/Song-Li/legacy_benchmark)

Dataset 3: Randomly selected packages

- **dataset:** The 500 randomly selected packages from the NPM database.
- **location:** We uploaded it as a zip file to the GitHub Repo (https://github.com/Song-Li/random_500_npm.git)

A.5.3 Play with the examples

In the ∼**/examples** folder of the Docker image, we have a few simple vulnerable examples for you to get familiar with our tool. You can try the **run_ipt.sh, run_os_command.sh** or **run_proto_pollution.sh** to run our tool on top of the pp_example.js(a prototype pollution) example and the motivating_example.js(the motivating example introduced in our paper). You can also write your modules, use a similar command and test them out.

A.6 Evaluation and expected results

A.6.1 Evaluation

Zero-day vulnerable packages detection (Dataset 1) Totally we have six different types of vulnerabilities, they are command injection, code execution, prototype pollution, path traversal, cross-site scripting, and internal property tampering. Each of them can be tested by running a command in the root directory of the source code:

- Command injection: ./scripts/os_command.sh
- Code execution: ./scripts/code_exec.sh
- Prototype pollution: ./scripts/prototype_pollution.sh
- Path traversal: ./scripts/path_traversal.sh
- Cross-site scripting: ./scripts/xss.sh
- Internal property tampering: ./scripts/ipt.sh

To reproduce the results, you can pick a vulnerability type and run the corresponding script.

Note that the scripts will try to run our tool parallelly, so you will not see the progress. Once you run a script, you should be able to see a message that says "new instance". You can check how many processes are still running by the command: *screen -ls*. You can also attach to a specific process by running: *screen -r XXX*(XXX means the name of the screen). Once all the processes are finished, you can check the result and run another script.

The testing results are located in the *logs* folder of the running directory. All the detected vulnerable packages will be output to the *succ.log* file; All the un-detected packages will be output to the *results.log* file. You can get the number of the successfully detected packages by running *cat ./logs/succ.log | wc -l*, during or after the running process.

If you finished checking one vulnerability type, please run *./clean.sh* to remove the logs and temporary files before checking another one.

Note that since the order of the testing functions is randomized, you may encounter some un-detected packages. For the un-detected packages, you may run them independently follow the instructions in README.md, or, go to *∼/packages/vulneralbility-type/package-name@version/* and run the *run.sh*

False negative rate (Dataset 2) The false-negative rate is introduced in Table 9 of the paper, which is measured on top of the legacy vulnerable packages. The steps to reproduce it is:

- Login to our Docker by the command *docker run -it iamthesong/odgen bash*
- Make sure you are in the root directory of the docker, and download the dataset by *git clone https://github.com/Song-Li/legacy_benchmark.git*
- Go into the downloaded dataset by *cd legacy_benchmark/* and unzip the dataset by *unzip legacy_benchmark.zip*
- Go into the source code directory by *cd /projs/ODGen/*. Test a type of vulnerability by *./odgen.py -t VUL_TYPE --list /root/legacy_benchmark/VUL_TYPE.list -aq --nodejs --timeout 120 --parallel 16*

Note that

- There are two locations in the last command that use *VUL_TYPE*. *VUL_TYPE* should be replaced by *os_command, ipt, proto_pollution, path_traversal* or *code_exec*
- The *--parallel* argument is used to run ODGen parallelly. In my case, I use 16 to indicate that I want to run 16 processes together. You can adjust the argument based on the number of CPU cores of your device
- The *--timeout* argument is used to set the timeout of a single test. We recommend 300 to make sure it works. In most cases, 120 should be enough.

If the number is less than expected, we need to run multiple times on those packages. You can simply run the same command multiple times (without cleaning the logs), and the results will be logged to the */root/projs/ODGen/logs/succ.log* file, cumulatively. To remove the duplicates, you can go into the */root/projs/ODGen/logs/* folder and run *awk '!x[$0]++' succ.log > outfile.succ*. The generated file *outfile.succ* will be the detected list.

Code coverage (Dataset 3) The code coverage rate is introduced in Figure 9 of the paper, which is measured on top of the 500 randomly selected Node.js packages. We prepared the statement-level code coverage API and the randomly selected 500 packages for testing.

Steps to reproduce:

Table 1: Expected Detection Results for Zero-day Vulnerable Packages

	Command Injection	Code Execution	Prototype Pollution	Path Traversal	Cross-site Scripting	IPT
#Packages	80	14	19	30	13	24
#Unpublished	2	4	0	0	0	0
#Expected	76∼78	9∼10	17∼19	30	12∼13	23∼24

Table 2: Expected True Positive Packages on Legacy Vulnerable Packages

	Command Injection	Code Execution	Prototype Pollution	Path Traversal	IPT	Total
#Packages	75	31	52	87	11	256
#Claimed True Positive	67	20	40	55	7	189
#Expected True Positive	67	20∼21	39∼40	55∼ 56	7∼ 10	189∼194

Table 3: Reproduced Code Coverage Rate

Code Coverage	Percentage of Packages
0% to 10%	5.52%
10% to 20%	5.25%
20% to 30%	8.01%
30% to 40%	2.76%
40% to 50%	2.76%
50% to 60%	6.63%
60% to 70%	2.76%
70% to 80%	6.35%
80% to 90%	11.60%
90% to 100%	48.34%

- Login to our Docker by the command *docker run -it iamthesong/odgen bash*
- Make sure you are in the root directory of the docker, and download the dataset by *git clone https://github.com/Song-Li/random_500_npm.git*
- Go into the downloaded dataset by *cd random_500_npm/* and unzip the dataset by *unzip ./random_500.zip*
- Go into the source code directory by *cd /projs/ODGen/*.
- Update the source code to the latest version by *git pull*
- Start the testing by running *./odgen.py -t os_command -ma –list /random_500_npm/random_500.list --timeout 30 --parallel 20*
- During the running process, you can go to the tools folder by *cd /root/projs/ODGen/tools* and check the results on the fly by running *python get_code_coverage_dis.py*. This script will output the results directly. You can run this command multiple times to see how the code coverage changes during the evaluation.

You can check how many processes are running by *screen -ls*. If all processes are finished, you can check the final result. Note that the code coverage raw data is logged in *ODGen/logs/stat.log*. You can also take a look if you want!

Note that not all of the packages will report code coverage, There are two reasons for that:

- Since the packages are randomly selected, there are many packages that do not meet the requirement of the NPM standard. For example, some of them do not have an entrance file, some of them do not include a package.json file, and some of them are demo packages without any meaningful content. For those packages, ODGen will not report the code coverage;
- It is possibly happening for packages running into a timeout. ODGen will not output the code coverage results for timeout packages since those results can not reflect the real code coverage of ODGen.

A.6.2 Expected results

Zero-day vulnerable packages detection The number of all the packages, unpublished packages, expected detected packages and the estimated running time are listed in Table 1

False negative rate The number of all the packages, claimed true positive packages, expected detected packages are listed in Table 2

Code coverage The distribution of the code coverage should be comparable to Figure 9 of the paper. The results that reproduced by the reviewers are listed in Table 3

A.7 Troubleshooting

Zero-day vulnerable packages detection If you can not get the expected results, you can try to restart the docker and see if it can run smoothly without the influence of the cache.

False negative rate If you can not get the expected results, you can try to:

- When you run the tool multiple times, try to change the number of *--parallel* each time. For example, we can use *--parallel 17* for the first time, and *--parallel 19* for the second time. In that way, each process may start from different packages and it may be faster to generate the results.
- Since the number of packages with code execution vulnerability is not very large. If your device has enough computing resources, for example, more than 20 CPU cores. You can try to set the *--parallel* argument to *--parallel 31* to make sure every vulnerable package can use an independent process. After doing this, you can check how many packages are still running by using *screen -ls*.

A.8 Experiment customization

You are very welcome to test our tool on top of your customized packages. To do so, please go to the ∼/example folder and write your package follow the NPM package standard, or write a module like the ∼/example/pp_example.js and the ∼/example/motivating_example.js.

Once you prepared the module, you can check out the README.md file in the source code repository and follow the instructions to run the corresponding commands.



A Artifact Appendix

A.1 Abstract

FUGIO is the first automatic exploit generation (AEG) tool for PHP object injection (POI) vulnerabilities. The artifact provides Docker images to reproduce the experiments performed in the paper. We tested these Docker images and scripts on a Ubuntu 18.04 machine. Each Docker container requires less than 5 GB of disk, but it requires more disk spaces for running FUGIO that stores identified POP chains and generates exploit objects to be fed to the fuzzer. We expect the artifact reproduces evaluations in Sections 7.2 and 7.3, producing Tables 1, 2, and 3 in the paper. Unfortunately, it might be hard to expect the same experimental results as the paper since FUGIO conducts fuzzing campaigns, and evaluations would be conducted in machines with different specifications.

A.2 Artifact check-list (meta-information)

- **Program:** We evaluated FUGIO on 30 PHP applications:
  - PHP 5.4: Contao CMS 3.2.4, Piwik 0.4.5, GLPI 0.83.9, Joomla 3.0.2, CubeCart 5.2.0, CMS Made Simple 1.11.9, Open Web Analytics 1.5.6, Vanilla Forums 2.0.18.5, SwiftMailer 5.0.1, SwiftMailer 5.1.0, Smarty 3.1.28, ZendFramework 1.12.20
  - PHP 5.6: PHPExcel 1.8.1, PHPExcel 1.8.2, Dompdf 0.8.0, Guzzle 6.0.0, WooCommerce 2.6.0, WooCommerce 3.4.0, Emailsubscribers 4.4.0, EverestForms 1.6.6 (w/ WordPress 5.0)
  - PHP 7.2: TCPDF 6.3.2, Drupal 7.78, SwiftMailer 5.4.12, SwiftMailer 6.0.0, Monolog 1.7.0, Monolog 1.18.0, Monolog 2.0.0, Laminas 2.11.2, Yii 1.1.20, TYPO3 9.3.0

  All benchmarks are included in the `benchmarks` directory. The artifact provides not only applications' source code also dump files of each application and its database for convenient settings.
- **Compilation:** FUGIO requires some libraries to be compiled. It needs only public compilers and the artifact provides all scripts to compile the libraries.
- **Run-time environment:** The artifact runs on Docker containers. We tested our Docker images and scripts on a Ubuntu 18.04 host machine. Given Docker images might work on any OS if it supports Docker.
- **Output:** After analyzing the target application, FUGIO generates 1) a PUT. When FUGIO identifies 2) a POP chain, it saves it as a file. If a POP chain reaches the sensitive sink, FUGIO reports the POP chain as 3) a probably exploitable chain. If the POP chain invokes the sink with a parameter containing the attack payload, FUGIO reports the POP chain as 4) an exploitable chain. All outputs are generated in the `Files/fuzzing/[APP_PATH.TIME]/PUT` directory.
  1. PUT: `put-head.php` and `put-body.php` are PUT files; `inst_PUT.php` is an instrumented PUT file for fuzzing the target application.
  2. POP chains: identified POP chains are stored as filename `procX_X_X_X_X_X.chain`.
  3. Probably exploitable chains: probably exploitable payloads are stored in the `PROBABLY_EXPLOITABLE` directory.
  4. Exploitable chains: exploitable payloads are stored in the `EXPLOITABLE` directory.

  During FUGIO identifies POP chains and generates their exploits, it periodically shows the progress to the console: how long FUGIO is running, how many POP chains are identified, how many POP chains are fed to the Fuzzer, how many probably exploitable payloads are generated, and how many exploitable payloads are generated.
- **Experiments:** We expect the artifact reproduces Tables 1, 2, and 3 in Sections 7.2 and 7.3. The artifact provides the `config.py` script for preparing the corresponding environment in which each experiment was conducted. However, it might be hard to expect the same experimental results since 1) FUGIO conducts fuzzing campaigns, which randomly produces results, and 2) evaluations would be conducted in machines with different specifications; our evaluations were performed on a Linux workstation equipped with 88 cores of CPUs and 384 GB of RAM.
- **How much disk space required (approximately)?:** Each Docker container does not require more than 5 GB of disk. However, FUGIO sometimes requires hundreds of GB depending on the target application since FUGIO identifies millions of POP chains.
- **How much time is needed to prepare workflow (approximately)?:** Preparing Docker containers and FUGIO takes less than an hour. Most of the time is spent on building the Docker image and installing dependencies.
- **How much time is needed to complete experiments (approximately)?:** The running time of FUGIO depends on the target application and the specification of the machine. For each application, Table 1 provides the time spent in running FUGIO on a machine equipped with 88 cores of CPUs. FUGIO can be run up to 12 hours for each target application.
- **Publicly available?:** The artifact is released at `https://github.com/WSP-LAB/FUGIO-artifact/tree/v0.1`.

A.3 Description

A.3.1 How to access

Users can access the artifact by cloning the repository from `https://github.com/WSP-LAB/FUGIO-artifact/tree/v0.1`.

A.3.2 Hardware dependencies

Although FUGIO does not require high-performance machines, it is better to have many cores of CPU and large capacities of RAM for parallel fuzzing. Note that we performed the experiments on a machine equipped with 88 cores of CPUs and 384 GB of RAM.

The artifact requires less than 5 GB of disk for each Docker container, but it requires more GBs for running FUGIO that stores identified POP chains and generates exploit objects to be fed to

the fuzzer. Depending on the target application, it might require hundreds GBs of disk space.

A.3.3 Software dependencies

We tested the Docker images and scripts on a Ubuntu 18.04 machine. The artifact requires only Docker; thus, it might work on any OS if it supports Docker. Other software packages will be installed using the provided scripts.

A.4 Installation

1. Install Docker and set that you can run docker commands with a non-root user.
2. Set up RabbitMQ by running the script `run_rabbitmq.sh`.
3. For each version of PHP, build Docker image using the script `1_docker_build.sh` and run a Docker container using the scripts `2_docker_run.sh` and `3_docker_exec.sh`.
4. In the Docker container, install dependencies for FUGIO by running the script `install_XX.sh`, depending on the version of PHP.
   - PHP 5.4: `install_54.sh`
   - PHP 5.6: `install_56.sh`
   - PHP 7.2: `install_72.sh`
5. Prepare environment for operating web applications. Start Apache web server and MySQL using the script `start.sh`. Then, make an account of MySQL using the script `create_user.sh`.

* For more details, please refer to the artifact repository.

A.5 Experiment workflow

1. Prepare a target web application. The artifact provides dump files of applications and databases for convenient settings. Install all or each application using the script `install.py`.
2. Add `.htaccess` file for monitoring POI vulnerabilities by running the script `htaccess.py`.
3. Prepare two terminals; one is for running FUGIO and the other is for triggering POI vulnerabilities.
4. In the first terminal, run FUGIO using the script `run_FUGIO_XX.sh` with the path of the target web application's source code.
5. In the other terminal, trigger the corresponding POI vulnerability using the given scripts in the `Trigger` directory.

* For more details, please refer to the artifact repository.

A.6 Evaluation and expected results

In the evaluations in Sections 7.2 and 7.3, we show that

1. FUGIO can automatically generate exploits for identified POP chains with zero false positives
2. FUGIO can generate exploits for some of the POP chains reported by Dahse *et al.*
3. FUGIO can generate new exploits compared to PHPGGC listed

using Tables 1, 2, and 3, respectively.

Table 1 shows that all exploitable chains that FUGIO generated are indeed exploitable chains (the left number of the plus sign in true positive chains). The number of true positive chains in Table 1 is manually analyzed. For Table 2, we could not match each exploitable chain since Dahse *et al.* did not provide the details of each chain. Thus, we compared the numbers of exploit objects that FUGIO reported with the numbers reported in their paper. Table 3 shows that FUGIO reported new 32 exploitable chains that PHPGGC does not list. PHPGGC provides templates for generating POP exploits. However, it is not clear that what POP gadgets each POP chain consists of. Thus, we provide POP chains from PHPGGC in the FUGIO repository (`https://github.com/WSP-LAB/FUGIO`). FUGIO repository also includes a utility for helping the analysis of the generated POP chains. For more details, please refer to the artifact repository.

The followings are steps for reproducing Tables 1, 2, and 3. First, please follow the installation step described in A.4. Second, for each target application, follow the instructions described in A.5. It takes much time to reproduce all target applications. We recommend selecting target applications that take less time and produce many exploits. When FUGIO finished analyzing the target application, FUGIO generates a dump file of summaries, which is stored in the directory `Files/dump_files`. When triggering a POI vulnerability of the target application using the given script, crawler, or other tools, FUGIO generates a PUT, which is stored in the directory `Files/fuzzing/[APP_PATH.TIME]/PUT`. When FUGIO identifies a POP chain, it saves it as a file in the same directory of the PUT file. If a POP chain reaches the sensitive sink, FUGIO reports the POP chain as a probably exploitable chain. If the POP chain invokes the sink with a parameter containing the attack payload, FUGIO reports the POP chain as an exploitable chain. Such explot objects are saved in the `PROBABLY_EXPLOITABLE` and `EXPLOITABLE` directories, respectively. FUGIO also prints the number of the identified POP chains, probably exploitable chains, and exploitable chains to the console. The results can be compared with Tables 1, 2, and 3 in the paper.



A Artifact Appendix

A.1 Abstract

TLS-Anvil is a test suite that evaluates the RFC compliance of Transport Layer Security (TLS) libraries using combinatorial testing (CT). To facilitate the automated analysis of multiple TLS libraries, we additionally provide the TLS-Docker-Lib project, which contains around 700 images for various versions of 22 TLS libraries. The test results composed by TLS-Anvil for the libraries BearSSL, BoringSSL, Botan, GnuTLS, LibreSSL, MatrixSSL, mbed TLS, NSS, OpenSSL, Rustls, s2n, tlslite-ng, and wolfSSL are the foundation of the evaluation in our paper. The results can be reproduced by running TLS-Anvil against local installations or docker images of the libraries. TLS-Anvil and its dependencies are written in Java, specifically for Java 11. Hardware requirements depend on the desired extent of the combinatorial testing (test strength). 16 GB of RAM are sufficient to test each library with strength three as we did for the paper.

A.2 Artifact check-list (meta-information)

- **Compilation:** TLS-Anvil and its dependencies can be built using Maven with Java 11.
- **Binary:** We provide executable jars built for Java 11 for TLS-Anvil here . TLS-Anvil is also provided as Docker image that is distributed using GitHub Packages. The TLS server/clients that we evaluated with TLS-Anvil are part of the TLS-Docker-Library. Those are designed as Docker images as well, but not available yet. However, the images can be built locally.
- **Data set:** We provide our raw test results to be used with the Report Analyzer here
- **Run-time environment:** We tested our artifacts using Linux or macOS. Since the TLS-Docker-Library contains some bash and python scripts this is also our recommendation. Those scripts depend on Docker. Therefore, root access is also needed.
- **Hardware:** No special Hardware needed. 16GB Ram is recommended to run TLS-Anvil.
- **Output:** The results of TLS-Anvil are stored in various json files. Those should be processed and evaluated with our Report Analyzer web application, which is also available as Docker image.
- **Experiments:** Analysis of TLS clients and servers using TLS-Anvil.
- **Required disk space:** Roughly 50 GB, considerably more if the whole Docker library is built (roughly 1 TB).
- **Approximate time required to prepare the workflow:** 2h
- **Time required to complete experiments:** Key results can be recreated quickly using a low testing strength of one, which takes around one hour for OpenSSL. Testing with strength two takes around six hours and strength three around 31 hours. While we evaluated with a strength of three, we identified that all findings can already be found using a strength of two. Generating all results for all libraries using docker images takes around one week when evaluating two libraries in parallel. Table 4 in our paper contains an overview of execution times.
- **Publicly available evolving repo:** The TLS-Anvil GitHub repo is available here.
- **Code licenses:** Apache 2
- **Data licenses:** Apache 2
- **Archived stable references:** The archived tags for TLS-Anvil, TLS-Docker-Library, and the Large-Scale-Evaluator are:
  - TLS-Anvil:
    Tag v1.0.3
  - TLS-Docker-Library:
    Tag 2.0.1
  - TLS-Anvil-Large-Scale-Evaluator:
    Tag 1.0.1

  The required versions of TLS-Anvil's dependencies are listed as submodules of the git repository.

A.3 Description

A.3.1 How to access

- TLS-Anvil can be found here.
- TLS-Docker-Library can be found here.
- TLS-Large-Scale-Evaluator can be found here.

Dependencies (optional)

- TLS-Attacker can be found here.
- TLS-Scanner can be found here.

See submodules of tags given in 'Archived' from subsection A.2 for specific versions.

A.3.2 Hardware dependencies

Depending on the desired testing strength, up to 16 GB of RAM are required. The overall CPU load of the system may affect the tests performed by TLS-Anvil. Please ensure that the system can provide enough processing resources for TLS-Anvil to obtain accurate results. A relatively recent CPU should be enough.

A.3.3 Software dependencies

To evaluate the considered libraries with TLS-Anvil and TLS-Docker-Lib, Java 11, Docker, Docker-Compose, and Maven are required. To build TLS-Anvil and its dependencies without Docker, Java 11 SDK is required. Running TLS-Anvil outside of a Docker container requires tcpdump.

A.3.4 Data sets

We provide the raw outputs of TLS-Anvil for the libraries considered in our evaluation. While we evaluated the libraries using a test strength of up to three, we identified that all findings can already be reproduced with a test strength of two. We hence provide the outputs for testing strength two.

A test output consists of multiple json files. These should be processed using our web application that visualizes the results (see below for more details).

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Setting up TLS-Anvil You can either use our provided Docker images or build everything yourself using a Dockerfiles contained in the TLS-Anvil repository. Note that the docker commands below fetch the latest version of TLS-Anvil.

a) Using the provided docker images:

1. Pull TLS-Anvil Docker image

```
docker pull \
ghcr.io/tls-attacker/tlsanvil:latest
```

2. Pull Report Analyzer Docker image

```
docker pull \
ghcr.io/tls-attacker/\
tlsanvil-reportanalyzer:latest
```

3. Pull Report Uploader Docker image

```
docker pull \
ghcr.io/tls-attacker/\
tlsanvil-result-uploader:latest
```

4. Adjust the tags to match a local build

```
docker tag \
ghcr.io/tls-attacker/tlsanvil:latest \
tlsanvil:latest
docker tag \
ghcr.io/tls-attacker/\
tlsanvil-reportanalyzer:latest \
uploader:latest
docker tag \
ghcr.io/tls-attacker/\
tlsanvil-reportanalyzer:latest \
reportanalyzer:latest
```

5. Clone the TLS-Anvil repository

```
git clone \
https://github.com/tls-attacker/\
TLS-Anvil.git
```

b) Building TLS-Anvil and the Report Analyzer yourself:

1. Clone the TLS-Anvil repository

```
git clone \
https://github.com/tls-attacker/\
TLS-Anvil.git
```

2. Run the build script

```
cd TLS-Anvil/
sh build.sh
```

3. Build the Report-Analyzer Docker image

```
cd Report-Analyzer/
docker-compose build
```

4. Build the upload Docker image (this uploads TLS-Anvil json files to the Report Analyzer web application)

```
cd src/backend/uploader
docker build -t uploader .
```

Setting up TLS-Docker-Library

1. Clone the repository using

```
git clone https://github.com/tls-attacker/\
TLS-Docker-Library.git
```

2. Navigate to TLS-Docker-Library/

3. Execute `setup.sh`

4. Execute `mvn install -DskipTests`

Downloading OpenSSL docker images

1. Get provided client and server images

```
docker pull ghcr.io/tls-attacker/\
openssl-client:1.1.1i
docker pull ghcr.io/tls-attacker/\
openssl-server:1.1.1i
```

2. Adjust the tags to match a local build

```
docker tag 4791200dbed9 \
openssl-server:1.1.1i
docker tag 8fe8f5106aa9 \
openssl-client:1.1.1i
```

Building an OpenSSL library Docker Container yourself (optional)

1. Navigate to TLS-Docker-Library/images/
2. Build the OpenSSL 1.1.1i server and client image

```
python3 build-everything.py -l \
openssl -v 1.1.1i
```

Building the TLS-Anvil-Large-Scale-Evaluator (optional)

1. Clone the repository

```
git clone https://github.com/tls-attacker/\
TLS-Anvil-Large-Scale-Evaluator.git
```

2. Navigate to TLS-Anvil-Large-Scale-Evaluator/
3. Run `mvn install -DskipTests`

A.5 Experiment workflow

TLS-Anvil can be run in client and server test mode depending on the tested endpoint. Regardless of the endpoint, TLS-Anvil first performs a feature discovery to determine suitable values for test parameters as well as applicable test templates. Since most test templates are either exclusively client or server test templates, many tests will be skipped during the execution. This is also the case for tests that must be skipped if an endpoint does not support a feature required for the test.

During the execution, TLS-Anvil creates json files for every executed test template containing the detailed results. The following guide shows how the OpenSSL server and client can be tested using TLS-Anvil. Both peers run inside a Docker container. The test results are stored inside the current working directory.

A.5.1 Testing OpenSSL Server

1. Create a separate Docker network

```
docker network create tls-anvil
```

2. Start the OpenSSL Server

```
docker run \
-d \
--rm \
--name openssl-server \
--network tls-anvil \
-v cert-data:/certs/ \
openssl-server:1.1.1i \
-port 8443 \
-cert /certs/rsa2048cert.pem \
-key /certs/rsa2048key.pem
```

3. Start TLS-Anvil

```
docker run \
--rm \
-it \
-v $(pwd):/output/ \
--name tls-anvil \
--network tls-anvil \
tlsanvil:latest \
-outputFolder ./ \
-parallelHandshakes 1 \
-strength 1 \
-identifier openssl-server \
server \
-connect openssl-server:8443 \
-doNotSendSNIExtension
```

A.5.2 Testing OpenSSL Client

1. Create a separate Docker network

```
docker network create tls-anvil
```

2. Start TLS-Anvil

```
docker run \
--rm \
-it \
-v $(pwd):/output/ \
--network tls-anvil \
--name tls-anvil \
tlsanvil:latest \
-outputFolder ./ \
-parallelHandshakes 3 \
-parallelTests 3 \
-strength 1 \
-identifier openssl-client \
client \
-port 8443 \
-triggerScript curl --connect-timeout 2
 ↪  openssl-client:8090/trigger
```

3. Start OpenSSL Client

```
docker run \
-d \
```

```
--rm \
--name openssl-client \
--network tls-anvil \
openssl-client:1.1.1i \
-connect tls-anvil:8443
```

A.5.3 Testing all Libraries

Since we needed to analyze multiple servers and clients of different libraries the manual process shown above results in a large overhead. Therefore, we automated the Docker container launching with a (Java) tool TLS-Anvil-Large-Scale-Evaluator. To analyze the OpenSSL server using this tool, the following command should be executed from the main folder of the cloned repository:

```
java -jar \
apps/TLS-Anvil-Large-Scale-Evaluator.jar \
-m server -e testsuite -i openssl -v 1.1.1i \
-p 1 -s 2
```

To analyze the client use:

```
java -jar \
apps/TLS-Anvil-Large-Scale-Evaluator.jar \
-m client -e testsuite -i openssl -v 1.1.1i \
-p 1 -s 2
```

After building the docker images for the libraries versions listed in the paper, you can run TLS-Anvil against all considered implementations using:

```
-i bearssl boringssl botan gnutls libressl \
mbedtls nss openssl rustls s2n tlslite_ng \
wolfssl matrixssl \
-v 0.6 3945 2.17.3 3.7.0 3.2.3 2.25.0 3.60 \
1.1.1i 0.19.0 0.10.24 0.8.0-alpha39 \
4.5.0-stable 4.3.0
```

A.6 Evaluation and Expected Results

To evaluate our results, you can either recreate them using the explanations from subsection A.5 or use the data from our experiments, which are available here.

A.6.1 Uploading Test Results to the Report Analyzer

To start with the evaluation you should start the Report Analyzer by navigating into the cloned TLS-Anvil repository and running:.

```
cd Report-Analyzer
docker-compose up -d
```

After that, a web application should be available at `http://localhost:5000`. The application offers three main pages: 'Upload', 'Analyzer', and 'Manage'. While it is possible to upload results using the web app, we recommend using the uploader Docker container as it collects all necessary files automatically. To start a recursive search for results from your current directory and upload them to the web application, use:

```
docker run \
--rm \
-it \
--network host \
-v $(pwd):/upload \
uploader
```

A.6.2 Analyzing Test Results

The drop down menu in the upper left corner of the 'Analyzer' page shows all test results uploaded to the Report Analyzer's database. In order to analyze a result, select a test result and click the 'Add' button next to the drop down menu. The Report Analyze will then show some basic execution properties, such as the execution time, and a list containing each test template that was executed and a symbol that indicates the test result. A check mark indicates a strictly succeeded test, a cross indicates a failed test. A check mark with a warning sign indicates a *conceptually* succeeded test, a cross with a warning sign indicates a *partially* failed test. An exclamation mark indicates further information is available on the test results.

Clicking on the result symbol leads to a detail page for the test template. The list shows the result for each test case executed throughout the test template. Clicking on an identifier on the left or a sub result on the right opens a modal window that summarizes the test inputs used for this specific test case as well as some meta data. Additionally, it is possible to inspect and download a pcap file that contains the recorded traffic of this specific test case.

Using the filter menu above the results table, it is possible to filter connections, for example, to only show connections where a parameter had a specific value or where a specific test result has been determined.

A.6.3 Key Results

Using TLS-Anvil, we identified a variety of RFC violations for the 13 considered libraries. Below, we describe how to identify some of the main findings using the Report Analyzer and our provided test results.

wolfSSL Authentication Bypass wolfSSL client 4.5.0 allows a server to bypass the authentication by sending a *Certificate* message with an empty certificate list. Open the test results for wolfSSL client (wolfssl-client-4.5.0-stable-WWkfM) in the Report Analyzer. Search for the

`emptyCertificateList` test in the list and click on the result symbol on the right. On the new page, some connections are marked as succeeded (with a checkmark), while others failed. The seemingly succeeded tests are the result of wolfSSLs intolerance for some record lengths. Since record fragmentation with different lengths is a parameter of our test input, wolfSSL sometimes can not finish the handshake. Since our RFC violation and the first fragmented records both take place within our first flight of messages, it appears as if wolfSSL sometimes correctly rejects our malformed *Certificate* message which is not the case. To obtain a clear test result, click on the drop down menu on the top of the page and select `RECORD_LENGTH`. This will filter the connections to only show a specific fragmentation length in bytes. Change the value on the right from 1 to 16384, which effectively shows only those handshakes where no record fragmentation was used. Inspecting the remaining test results of the list, either by hovering on the result symbol or clicking on it, shows that wolfSSL always failed to reject the invalid message and instead proceeded to send its final handshake message and application data. You can compare this behavior to the very similar `emptyCertificateMessage` test, where wolfSSL behaves as expected. In contrast to the previous test, here we use an entirely empty message (with a message length of zero). By applying the same filter steps as before, the results show that wolfSSL rejects `this type` of empty *Certificate* message correctly.

MatrixSSL Padding Oracle Vulnerability The MatrixSSL 4.3.0 client indicates an invalid padding upon decryption for ciphersuites that use SHA256 to compute the HMAC. Open the same test result as before and search for the test `invalidCBCPadding`. MatrixSSL aborted the connection for all messages that contained an invalid padding value. However, for SHA256 cipher suites, MatrixSSL does not send an alert as it does for all other cipher suites. We further analyzed this behavior and identified that this is due to a segmentation fault. This behavior is unique to the invalid padding error case and thus leaks information about the obtained plaintext. You can compare this behavior to the `invalidMAC` test, where MatrixSSL always sends an alert regardless of the cipher suite.

MatrixSSL Unproposed Groups The MatrixSSL 4.3.0 client accepts that a server negotiates certain curves that have not been proposed by the client. While MatrixSSL offers the curves secp256r1, secp384r1, x25519, and secp521r1 it also accepts *ServerKeyExchange* messages that contain a public key of the curves secp192r1 and secp224r1, which both have significantly weaker security properties. To identify this behavior, open the test results for MatrixSSL client (matrixssl-client-4.3.0-ik8fF) and search for the test `acceptsUnproposedNamedGroup` and click on the test result symbol. Using the drop down menu at the top, select 'Test Result' as the filter and set the desired value to 'FAILED' in the drop down menu on the right. By clicking on the remaining test results, the Report Analyzer shows a text box with a summary of details in json. First of all, the `DerivationContainer` element shows the chosen parameters of the test. The `NAMED GROUP` parameter for the failing tests is either secp192r1 or secp224r1. Further below, the `Stacktrace` shows the failed JUnit Assertion with an indication of the error. In this case, an alert was expected (since the server made an illegal selection) but MatrixSSL client proceeded to send a *ClientKeyExchange*, *ChangeCipherSpec*, and *Finished* message instead.

A.7 Experiment customization

You can also run your own experiments with TLS-Anvil against any server or client. For this purpose run the jar available in TLS-Anvil/TLS-Testsuite/apps from the cloned and built repository or use our provided jars. To test a server running on localhost:4433, use:

```
java -jar TLS-Testsuite.jar server -connect \
localhost:4433
```

To test a client from port 4433, use:

```
java -jar TLS-Testsuite.jar client -port 4433 \
-triggerScript triggerScript.sh
```

Where `triggerScript.sh` contains the command to start a client that connects to localhost:4433.

A.8 Notes

Analyzing the issues for a scientific paper sometimes required additional manual labour, as we grouped failed tests based on their route cause to get to the final number of findings. Therefore the number of failed tests is larger than the number of findings.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

Keybuster is a research tool that allows to interact with the Keymaster TA (Trusted Application) on Samsung devices that run Android. Keybuster implements a Keymaster client - based on the `libkeymaster_helper.so` library from Samsung's Keymaster HAL.

Keybuster requires sufficient permissions (root and SELinux context) to access TZ drivers. To achieve this, we rooted our device using Magisk and used the strong context that it provides. Keybuster requires special hardware - an Samsung Galaxy smartphone (S9 and newer models) with a Trusted Execution Environment (TEE). The binary can be downloaded from GitHub releases or built using Android NDK. To reproduce our attacks, only minimal software requirements are required (e.g., adb/ssh, openssl and python3).

Keybuster allows to reproduce the attacks that we describe in the paper - the IV reuse attack and the downgrade attack. The GitHub repository contains detailed step-by-step instruction on how to recreate both attacks. Additionally, it allows researchers to freely explore the Keymaster TA without input validation or filtering. Samsung validated both attacks using Keybuster and assigned a High severity CVE to each issue.

In essence, the proof-of-concept attacks utilize Keybuster to demonstrate private key material extraction of hardware-protected keys that were encrypted by the TEE.

A.2 Artifact check-list (meta-information)

- **Compilation: Android NDK (alternatively, the binary can be downloaded from GitHub releases)**
- **Binary: keybuster binary for Android, included in GitHub releases**
- **Run-time environment: Android specific, requires a sufficiently strong context (e.g., rooted device)**
- **Hardware: Rooted Samsung Galaxy device (e.g. available over adb/SSH)**
- **Security, privacy, and ethical concerns: The vulnerabilities were responsibly disclosed to Samsung and they issued patches. Running on a rooted Android device (that is connected to WiFi) over SSH might be dangerous.**
- **Output: Console output. GitHub repository includes expected output.**
- **Experiments: Follow instructions in GitHub repository to run the proof-of-concept scripts**
- **How much disk space required (approximately)?: Minimal**
- **How much time is needed to prepare workflow (approximately)?: < 2 minutes**
- **How much time is needed to complete experiments (approximately)?: < 2 minutes**
- **Publicly available (explicitly provide evolving version reference)?: Will be made available on `https://github.com/shakevsky/keybuster`**
- **Code licenses (if publicly available)?: Apache-2.0 License**
- **Archived (explicitly provide DOI or stable reference)?: Will be made available on `https://github.com/shakevsky/keybuster/tree/v0.1.0`**

A.3 Description

A.3.1 How to access

The stable artifact will be made available on `https://github.com/shakevsky/keybuster/tree/v0.1.0`.

A.3.2 Hardware dependencies

To reproduce our attacks with Keybuster it is required to have a Samsung device (S9 and newer models) with a sufficiently strong context, e.g., by rooting a device or by having a development device from Samsung. We can try to make such device available over SSH. Unpacking the artifact requires very little space (only to download the binary or compile the sources).

A.3.3 Software dependencies

Keybuster is a binary that can be run on a rooted Samsung device. To access such a device we've used adb, and we can try to make a vulnerable device available over SSH. To reproduce the IV reuse attack, no additional software is required. To reproduce the downgrade attack (e.g., against a simplified Secure Key Import server) python3 and openssh can be used (although they only emphasize the point - running the proof of concept script should be enough).

A.3.4 Security, privacy, and ethical concerns

The vulnerabilities were responsibly disclosed to Samsung and they issued patches. We have some concerns over giving SSH access to a rooted Android device over the internet, as it can potentially compromise the home WiFi network of one of the authors.

A.4 Installation

Assuming that we'll provide remote SSH access, we can upload all the necessary files to the device so that no further setup is required and reviewers can simply run the proof-of-concept scripts. Otherwise, the GitHub repository includes detailed steps of how to reproduce the attacks (which bash commands to run).

A.5 Evaluation and expected results

Full details on reproducing the proof-of-concept attacks, as well as expected outputs, will be made available in the GitHub repository.

The main claims of our paper include:

- We show that the hardware protection in Samsung Galaxy S9 devices is vulnerable to an IV reuse attack on AES-GCM, allowing the extraction of protected key material.

- We show a downgrade attack on Samsung Galaxy S10, S20, and S21 devices, making them vulnerable to our IV reuse attack.
- We evaluate the impact of our attacks and describe how to exploit them to misuse the Keystore key attestation to bypass FIDO2 WebAuthn login and compromise Google's Secure Key Import.

The proof-of-concept attacks that use Keybuster support the claims:

- The IV reuse PoC shows that we are able to fully recover key material from hardware-protected keys that were encrypted by the TEE.
- The downgrade attack PoC shows that we are able to force even the latest devices (S10, S20, and S21) to generate keys that are vulnerable to IV reuse.
- The README.md of the downgrade attack also includes information about how an attacker can use the downgrade attack to target higher level cryptographic protocols such as Secure Key Import (we included python3 code that emulates the server and show how a successful attack breaks the security of the keys) and WebAuthn (we included a GDB script that we've successfully used against the StrongKey FIDO2 WebAuthn server, as described in the paper).

The expected outputs and results are shown in the proof-of-concept README files in the GitHub repository.

A.6 Notes

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

To examine the threats perceived and faced by migrant domestic workers (MDWs) to their security and privacy, we designed and conducted five participatory threat modelling workshops with MDWs in the UK. Drawing on the findings of our workshops, we created and disseminated a free online digital privacy and security guide (hosted on GitHub), to make our research outputs accessible to the public as well as organisations that protect migrant and precarious workers in the UK[1]. We developed the guide in collaboration with Voice of Domestic Workers (VoDW), an education and support group run by and for migrant domestic workers, and Migrants Organise, a grassroots platform where migrants and refugees organise for justice. During each workshop, we asked participants whether they had any questions for us on online safety, privacy, and security. We noted down these questions as well as the threats participants had identified and their advice for other MDWs. We then used these as the basis of our online digital privacy and security guide. In making the guide, we focused, where possible, on existing resources, such as the DIY Guide to Feminist Cybersecurity[2], the Citizens Advice online scams helper[3], and Kalayaan's Employment Rights webpage[4]. We made sure to include clear action points the reader could easily implement. We also focused on making the guide easily readable. Lastly, we avoided unnecessary intimidation or victim blaming. For example, we included reminders like "Avoiding surveillance by your employer should not have to be your responsibility. Employers need to understand and respect domestic workers' right to privacy and safety, and refrain from excessive monitoring". To refine and validate the guide, we are continuously soliciting feedback from our computer privacy and security as well as MDW communities on the guide, are incorporating feedback on a regular basis, and plan to organise workshops with MDWs to hear their input on the guide. We also aim to translate our guide from English into other languages. The appendix of our USENIX Security paper describes our guide structure in detail (see Appendix B in the main paper).

A.2 Artifact check-list (meta-information)

- **Security, privacy, and ethical concerns:** Ethical considerations for this study included preserving the anonymity of vulnerable participants. We followed principles of data minimisation, ensuring that the research data that we collected was not connected to participants' identities. In order to do this, we did not video or audio record workshops; we instead relied on handwritten notes which did not include participants' names, as well as used an online platform where participants could submit answers to our questions anonymously. Some participants also participated in our study outside their home and workplace; e.g., in a park, in order to avoid being overheard by employers. Further, the only researchers with access to the personal/contact details of participants were those being involved in data collection and analysis. We also note that although some participants had experiences of being undocumented in the past (as a result of recruitment by our peer researcher), all participants had right to remain at the time of the study.

  Another concern was the potential distress of participants who discussed difficult or sensitive experiences. To mitigate this, we reminded participants that they did not need to answer the questions, and they could take breaks during the study. Further, our peer researcher at VoDW was present at all workshops to make sure that participants felt comfortable.

  Lastly, our research was reciprocal, to make sure participants benefited from the project, particularly as they belonged to a vulnerable group in often precarious employment. We compensated each participant £50, and we attempted to ensure the accessibility of our research outputs through publishing a digital privacy and security guide online. This study was approved by the Research Ethics Committee at the University of Oxford.
- **Output:** We attempted to ensure the accessibility of our research outputs through publishing a digital privacy and security guide online on GitHub: https://github.com/domesticworkerprivacy/domesticworkerprivacy.github.io/tree/33fc93f2a192378180a5f6eb235f384d07c67ced.
- **Experiments:** The appendix of our USENIX Security paper describes the questionnaire we used during our workshops, which can be used in similar future studies in different countries (see Appendix A in the main paper).

  The topics discussed in the workshops were as follows:

  ASSETS:

  1. What kinds of social media or communication technology do you use?
  2. How do you feel about using social media and communication technology?
  3. What parts of your data or information do you most want to protect?
  4. What does being safe mean to you?

  THREATS:

  1. What are the main threats to your safety, privacy, and security (e.g., threats faced online or in your workplace)?
  2. Have you ever worked in a house where there was a camera or some type of a monitoring device? If yes, how did you find out about it? How did you feel about it?
  3. Are you worried about being watched online? If so, by who and why?

  MITIGATIONS:

[1] Accessible here: https://domesticworkerprivacy.github.io/

[2] https://hackblossom.org/cybersecurity/

[3] https://www.citizensadvice.org.uk/consumer/scams/what-to-do-if-youve-been-scammed/

[4] http://www.kalayaan.org.uk/for-workers/employment-rights/

1. What advice would you give other MDWs to stay safe online?
2. What parts of your safety do you most want to improve?
3. Do you have any questions you want to ask us?
4. What kind of support do you need to be safe?

- **Publicly available (explicitly provide evolving version reference)?:** The first version of the guide is accessible here: https://github.com/domesticworkerprivacy/domesticworkerprivacy.github.io/tree/33fc93f2a192378180a5f6eb235f384d07c67ced.
- **Code licenses (if publicly available)?:** The first version of the guide is accessible here: https://github.com/domesticworkerprivacy/domesticworkerprivacy.github.io/tree/33fc93f2a192378180a5f6eb235f384d07c67ced.
- **Archived (explicitly provide DOI or stable reference)?:** The first version of the guide is accessible here: https://github.com/domesticworkerprivacy/domesticworkerprivacy.github.io/tree/33fc93f2a192378180a5f6eb235f384d07c67ced. No changes have been made.

A.3 Description

A.3.1 How to access

The first version of the guide is accessible on GitHub: https://github.com/domesticworkerprivacy/domesticworkerprivacy.github.io/tree/33fc93f2a192378180a5f6eb235f384d07c67ced. No changes have been made.

A.3.2 Hardware dependencies

N/A.

A.3.3 Software dependencies

N/A.

A.3.4 Data sets

N/A.

A.3.5 Models

N/A.

A.3.6 Security, privacy, and ethical concerns

Please see above in §A.2.

A.4 Installation

Drawing on the findings of our study, we created and disseminated a free online digital privacy and security guide (hosted on GitHub), to make our research outputs accessible to the public as well as organisations that protect migrant and precarious workers in the UK[5]. Our guide is publicly accessible to everyone.

A.5 Experiment workflow

N/A.

A.6 Evaluation and expected results

Our artifact is a digital privacy and security guide. The guide serves as an educational/support platform for MDWs in the UK and other countries, to protect their on- and offline privacy and keep themselves safe. The guide is divided into six main sections. We first explain the guide and its purpose; provide general digital privacy and security advice; describe three main types of privacy threats identified by our MDW participants who took part in our workshops (one section is dedicated to each threat type): government surveillance, online scams and harassment, and employer monitoring; and conclude by arguing that our computer security and privacy community must take into account intersecting forms of marginalisation (due in part to different levels of social and economic power) as well as the broader social structures that foster insecurity. The guide also includes links to further resources that domestic workers can refer to when in need of protection; see the appendix of the main paper for more details.

To develop similar guides with MDW communities or validate our guide, researchers can recruit MDW participants in different countries – bearing in mind the ethical considerations we described above – as well as use the questionnaire we provided above in §A.2.

A.7 Experiment customization

N/A.

A.8 Notes

N/A.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

[5] Accessible here: https://domesticworkerprivacy.github.io/



A Artifact Appendix

A.1 Abstract

This artifact includes software that covers all 3 basic functionalities needed to reproduce the results of our paper "How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes". The three main functionalities are (a) data collection, (b) heuristic execution (code analysis), and (c) experiments (analysis incl. plots, tables, etc. that are presented in the paper). The artifact shows that the process described in the paper is reproducible and the results of executing the artifact (tables, plots) should be similar to the results reported in the paper (by executing the artifact now, new CVEs will be added to the analysis so results are expected to differ slightly from the ones reported in the paper).

The artifact is shipped as a docker image (the repository includes a Dockerfile that can be used to create the image). We tested on Docker version 18.09.1 on Debian GNU/Linux 10. Function (b) "heuristic execution" is CPU-intensive and parallelized, so we recommend using a machine with many CPU cores to speed up the process. The docker container requires ~60GB of disk space, mainly to download the repositories of the projects in the study. Although we tested the artifact on a "big" 128-core machine, we expect it to run without problems on "smaller" machines. We offered reviewer of the AE Committee of USENIX Security '22 ssh access to the machine we used for testing.

A.2 Artifact check-list (meta-information)

- **Data set:** The "ground-truth" dataset (CVE to VCC mappings) is included as a set of files in the repository. The process to create the main dataset is performed by the artifact.
- **Run-time environment:** The artifact is intended to be executed in a docker container, so docker is required (which generally also implies root privileges on the machine).
- **Hardware:** No specific hardware is required, although the execution of the experiments can be accelerated with the use of multiple CPU cores (heuristic execution) and good bandwidth (cloning repositories). Since the execution can take long to complete, a dedicated machine or server (with internet access) is required.
- **Output:** The output consists of the tables and plots included in the paper.
- **Experiments:** The artifact includes a "run_all.sh" bash script that performs all the actions required. Alternatively the user can run the commands in this script manually.
- **How much disk space required (approximately)?:** 60GB
- **How much time is needed to prepare workflow (approximately)?:** 1-5 min.
- **How much time is needed to complete experiments (approximately)?:** 10-80 hrs (mainly depending on number of cores available and bandwidth)
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/manuelbrack/VulnerabilityLifetimes/tree/usenix_ae
- **Code licenses (if publicly available)?:** GPL-3.0
- **Data licenses (if publicly available)?:** CC BY 4.0
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/manuelbrack/VulnerabilityLifetimes/tree/usenix_v1.0

A.3 Description

A.3.1 How to access

https://github.com/manuelbrack/VulnerabilityLifetimes/tree/usenix_ae

A.3.2 Hardware dependencies

The process requires ~60GB of disk space. This is mainly because the artifact looks into the repositories of some big projects, such as chromium and Linux. More cores will mean the artifact will run faster.

A.3.3 Software dependencies

Git to clone the repository and docker to build and run the image.

A.3.4 Data sets

The required datasets are either included in the repository or created by the artifact.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Clone the repository[1] and follow the instructions in the readme[2] to build and run the docker image (docker usually implies that root access is required on the machine you are using).

[1] `git clone --branch usenix_v1.0` https://github.com/manuelbrack/VulnerabilityLifetimes

[2] https://github.com/manuelbrack/VulnerabilityLifetimes/blob/usenix_ae/README.md

A.5 Experiment workflow

The artifact implements the 3 main functionalities required to reproduce the results of our paper: (a) dataset creation, (b) heuristic execution, and (c) results analysis. All required steps are included in a "run_all.sh" bash script with comments that explain the purpose of each step. At the end we recommend that you copy the contents of the "/project/VulnerabilityLifetimes/out/" directory[3] to a machine with a GUI so you can inspect the plots.

A.6 Evaluation and expected results

Note that this is a measurement paper and that the artifact also implements the critical part of dataset creation. The ability of this code to create an up-to-date dataset automatically is a key contribution of this artifact. Given that the dataset collection for the paper was executed some months before publication, you should expect some variation of the results caused by new data points. You can get exactly the same results as the ones reported in the paper by importing the mappings from https://figshare.com/s/4dd1130c336f43f6e18c and running the analysis scripts but there is no real reproduction value there. The main claims of the paper can be summarized by the following:

1. The heuristic provides good estimates for vulnerability lifetimes
    - Check the output of the `/out/heuristic.csv` file for the content of Table 2 of the paper. Here you should note that the numbers reported are similar; especially, the numbers in the last 2 columns of the file are smaller than the respective numbers of the previous columns.
    - Check the plots under `./out/year_trends/year_trend_linux_gt_comp.pdf`, `./out/distributions/distribution_gtdata_gt.pdf`, `./out/distributions/distribution_gtdata_heuristic.pdf`, as well as the qqplots in the same directory. They should be similar to Figures 3 and 4, showing that the results of the heuristic are close to the ground truth data.
2. *Section 5.1:* Look into `./out/lifetimes_table.csv` for results similar to the ones reported in Table 3. You should be able to observe big differences between projects and a higher average/mean value than median. This table can also be used as a check to see if the experiment has been executed successfully. If results in your file are similar to the ones reported in the paper, then you can be pretty sure that the experiment ran correctly.
3. *Section 5.2:* Look into the plots at `./out/distributions/distribution_All_pdf.pdf` and `./out/distributions/qq_plot.pdf`. for similar results to Figures 5 and 6. Here you should be able to observe that the exponential distribution is a good fit to the data.
4. *Section 5.3:* Inspect plots with the naming convention `./out/year_trends/year_trend_{project}.pdf` for similar results to Figure 7 (increasing trends except for Firefox).
5. *Section 5.4:* Look into the plots at the directory `./out/regular_code_age/` for similar results to Figure 9. Here you should be able to observe the correlation between vulnerability lifetimes and code age and, especially for Chromium, that lifetimes are increasing slower than code age.
6. *Section 5.6:* Look at the plot at `./out/year_trends/year_trend_kernel_mem_vs_others.pdf` for similar results to Figure 10. You should be able to note that the trend is increasing both for memory vulnerabilities and for other types.

Apart from the main results listed above, you can find many more results (both presented in the paper and additional material in the directories referenced above). Also, some results, such as the statistical tests for vulnerability types are printed in stdout. You can find these results in the log file of the execution.

A.7 Experiment customization

The code is written in a way that new projects and data sources can be added with relatively little additional effort (although not trivially). See the readme in the main branch of the repository for more information. However, the scripts for the artifact evaluation do not support seamless addition of projects to analyze. This could be a point for future work.

A.8 Notes

Warnings during the execution of the artifacts are not suppressed and are to be expected. Here is a short explanation:

- 'Cannot add or update a child row...': a fixing commit-CVE mapping has been identified by text mining techniques but the CVE is not in the list of CVEs that affect the project as identified by cpe.
- 'CVE search: 89it [00:14, 37.71it/s]...': The CVE entry is not complete.
- '62f4f82ad39f177538f733b37cdd5dabd8f333de could not be saved...': Commit message includes a picture (emoji) – see https://github.com/chromium/chromium/commit/62f4f82ad39f177538f733b37cdd5dabd8f333de. This can be fixed in a future version of the tool.

[3] e.g. `docker cp` and then `scp` if you are connected to a server

- ‘warnings.warn("Commit not found 0".format(commitsha))‘: a fixing commit-CVE mapping has been identified by text mining techniques but the commit is not in the repository.
- ‘ValueWarning: omni_normtest‘: some projects have few points and such warnings are natural.
- ‘UserWarning: no blames‘: No commits were blamed by the heuristic for a given fixing commit, e.g. because it changed only non C/C++ files.
- ‘WARNING:root:SKIPPED powernorm distribution (taking more than 30 seconds)‘: Expected warning from the fitter package (https://fitter.readthedocs.io/en/latest/)

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

We developed a Web platform and an API client that allow users to retrieve the Expected Exploitability (EE) scores predicted by our system. The system gets updated daily with the newest scores.

We implemented an API client in python, distributed via Jupyter notebooks in a Docker container, which allows users to interact with the API and download the EE scores to reproduce the main result from the paper, in **Figure 5(a)** and **Figure 8(a)**, or explore the performance of the latest model and compare it to the performance of the models from the paper.

A.2 Artifact check-list (meta-information)

- **Program:** Docker (tested on v20.10.8)
- **Run-time environment:** UNIX-like system
- **Metrics:** Rrecision, Recall, Precision-Recall AUC, TPR, FPR, AUC.
- **Output:** Plots from the paper, reproducing and expanding Figure 5(a) Figure 8(a).
- **Experiments:** Running Jupyter notebooks.
- **How much disk space required (approximately)?:** 4GB
- **How much time is needed to prepare workflow (approximately)?:** 15 min
- **How much time is needed to complete experiments (approximately)?:** 30 min
- **Publicly available?:** The website and client code are publicly available. The API requires an API token which we provide upon request.

A.3 Description

A.3.1 Web Platform

The Web platform exposes the scores of the most recent model, and offers two tools for practitioners to integrate EE in vulnerability or risk management workflows.

The *Vulnerability Explorer* tool allows users to search and investigate basic characteristics of any vulnerability on our platform, the historical scores for that vulnerability as well as a sample of the artifacts used in computing its EE. One use-case for this tool is the investigation of critical vulnerabilities, as discussed in Section 7.3 - EE for critical vulnerabilities.

The *Score Comparison* tool allows users to compare the scores across subsets of vulnerabilities of interest. Vulnerabilities can be filtered based on the publication date, type, targeted product or affected vendor. The results are displayed in a tabular form, where users can rank vulnerabilities according to various criteria of interest (e.g., the latest or maximum EE score, the score percentile among selected vulnerabilities, whether an exploit was observed etc.). One use-case for the tool is the discovery of critical vulnerabilities that need to be prioritized soon or for which exploitation is imminent, as discussed in Section 7.3 - EE for emergency response.

A.3.2 API Client

The API allows clients to download historical scores for a given vulnerability (using the `/scores/cveid` endpoint), or all the prediction scores for a particular model on a particular date (using the `/scores/daily` endpoint). The API documentation describes the endpoints and the parameters required for each call, and provides example code for clients to interact with the API.

A.4 How to access

The Web platform is available at https://exploitability.app/. The API and the client code are available at https://api.exploitability.app/.

A.5 Installation

The practitioner tools are available on the Web platform. To use the API, clone the code repository, point a terminal to that folder and run `bash docker/run.sh`. This will create the Docker container and spawn a Jupyter server. Use the URL displayed in the console to open a browser session within that container.

A.6 Evaluation and expected results

To reproduce the results from the paper, open and run the following notebook: `reproducibility_plot_performance.ipynb` In an API key is provided, this will download the required scores used in the paper, cache them in various files in `scores_reproducibility_download/`, and use these files to compute the performance of EE and baselines. The output consists of 2 figures, which correspond to Figure 8(a) and Figure 5(a) in our paper.

To evaluate the latest model, open and run the following notebook: `latest_plot_performance.ipynb` In an API key is provided, the notebook will download all the scores produced by our latest model on `2021-10-10`, cache them into a file in `scores_latest_download/`, and use this file to compute the performance of EE. The output consists of 2 figures which are comparable to Figure 8(a) and Figure 5(a) in our paper.

As of December 2021 we observe that the performance of our latest model predicting EE, computed before `2021-10-10`, is very close to the performance reported in the paper (0.69 PR AUC / 0.96 AUC for latest model vs 0.73 PR AUC / 0.84 AUC in the paper), demonstrating that our online predictor is functional and in line with the claims from the paper. The performance of the latest model on other dates can be computed by changing the `SCORES_DATE` variable and re-running the notebook.

A.7 Experiment customization

When running `latest_plot_performance.ipynb`, users can customize the `SCORES_DATE` variable in the notebook to observe the performance of our model on different dates.



A Artifact Appendix

A.1 Abstract

The artifact provided is the implementation of ARBITER framework along with the VD implementations for 4 CWE types and for the Juliet data set. The framework, as well as, the VD templates are written in Python. The artifact contains a helper script written in Python that invokes the Arbiter API when provided with a VD template on a target binary both of which can be specified via command-line arguments. The artifact requires machines that contain 1 logical core and at least 4 GB of RAM. A containerized copy of the artifact is available and can be used on any systems that support docker. The software requirements for installing are Python (version at least 3.8) and *angr*. The artifact has been tested on a machine running Ubuntu 18.04. The artifact also contains the list of packages that were used for evaluation, a JSON file containing the MD5 hashes of each of the binaries as well as the actual binaries from the Juliet data set that were evaluated.

Our paper describes ARBITER as a combination of static analysis and dynamic symbolic execution that can be used to detect classes of vulnerabilities in binary programs with high scalability and low false positive rate. The large scale evaluation on 76,516 x86-64 binaries in Ubuntu repositories as well as the evaluation on Juliet Test Suite (v1.3) highlight these properties of ARBITER .

In order to validate the experiments, one can repeat them using the provided templates and compare the results with those presented in the paper. Since the underlying techniques used by ARBITER contain a degree of non-determinism, the results may vary slightly when evaluating on larger binaries. However, the overall results will be comparable to those presented in the paper.

A.2 Artifact check-list (meta-information)

- **Binary:** Binary executables from the Juliet Test Suite (v1.3) that were used during the evaluation are included in the artifact.
- **Data set:** The artifact contains a list of packages from the Ubuntu repository and a JSON file that contains the MD5 sums of each binary used for the evaluation.
- **Run-time environment:** The artifact was verified to work on Ubuntu 18.04 and requires Python (version at least 3.8) and *angr* binary analysis framework.
- **Hardware:** Our experiments were performed on a kubernetes cluster where each pod was provided 1 logical core and 4 GB of RAM. However, each pod could request up to 8 GB of RAM.
- **Execution:** The artifact contains a helper script that can be executed using the Python interpreter. The arguments to this script include the VD template to use as well as the target binary to analyze.
- **Metrics:** The metric used in the experiment is the false positive rate of the bugs reported.
- **Output:** Each template outputs the bugs that it detects in the target binary. The results are also stored into log files and json files that are saved on the disk.
- **Experiments:** To prepare and run the experiments, steps required are as follows.
  1. Download the relevant binary if required.
  2. Clone the artifact repository from https://github.com/jkrshnmenon/arbiter and install it or pull the docker image 4rbit3r/arbiter:latest.
  3. Execute the helper script named *run_arbiter.py* provided in vuln_templates/ with a VD template and a path to the target binary as arguments.
- **How much disk space required (approximately)?:** The total disk space used, including downloaded binaries and generated output files, for our experiment is approximately 350 GB.
- **How much time is needed to prepare workflow (approximately)?:** Installing the framework or using the docker container should take nearly 5 minutes. However, downloading the binaries could take up to 1 minute per package. Even if this process is performed in parallel, it could take up to 1 hour to download all the packages depending upon the network speed.
- **How much time is needed to complete experiments (approximately)?:** Our evaluation was performed on a kubernetes cluster that allowed running 800 tasks at a time. With that constraint, our evaluation of 76,516 binaries took nearly 2 days to complete per template.
- **Publicly available (explicitly provide evolving version reference)?:** The artifact is available at https://github.com/jkrshnmenon/arbiter/releases/tag/v1.1 and as a docker image 4rbit3r/arbiter:latest
- **Workflow frameworks used?:** We used kubernetes in our experiments that allowed us to run 800 tasks at a time with each task being allotted 1 logical core and 4 GB of RAM.

A.3 Description

A.3.1 How to access

The artifact is publicly available at https://github.com/jkrshnmenon/arbiter/releases/tag/v1.1 and as a docker image 4rbit3r/arbiter:latest.

A.3.2 Hardware dependencies

The artifact requires 1 logical core and at least 4 GB of RAM.

A.3.3 Software dependencies

The artifact has been verified to work on Ubuntu 18.04 and requires Python (at least version 3.8) and the *angr* python package.

A.3.4 Data sets

A list of the packages used and JSON file containing the binaries and their MD5 sums are provided. The corresponding binaries can be downloaded from the Ubuntu repositories. The binaries from the Juliet Test Suite (v1.3) are provided with the artifact.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

The artifact contains a setup script that can be executed to install the framework. After the repository has been cloned, the following command can be used to install the framework : *python setup.py install*.

A.5 Experiment workflow

The experiment was performed on a kubernetes cluster. In order to deploy tasks on the cluster, a docker image has to be specified. We create docker images for each CWE type that would execute the corresponding template script against a specified target binary and wrote the results to disk. This process was repeated for each CWE type using the corresponding template script.

Since the Juliet data-set only provides documentation about the locations of vulnerabilities in terms of source code files and line numbers, a mapping between this location and the address of corresponding function in the compiled binary is required.

ARBITER provides the function address where the vulnerability has been detected. This information, combined with the ground-truth from the Juliet data-set can be used to evaluate the false-positive rate of ARBITER .

A.6 Evaluation and expected results

The paper highlights the high scalability and low false positive rates of ARBITER . The key results that highlight these properties are :

- The large scale evaluation on 76,516 x86-64 binaries on 4 different CWE types.
- The resulting reports that were manually evaluated and the false positive rate was determined to be nearly 40%.
- The evaluation on the Juliet Test Suite (v1.3) where the false positive rate was found to be nearly 23%.

In order to reproduce these results, the templates can be evaluated against the target binaries and the results generated can be manually verified. The expected false positive rate across all the 4 CWE types on the entire data set of 76,516 x86-64 binaries in the Ubuntu repositories is nearly 40% while the expected false positive rate for the binaries from the Juliet Test Suite is nearly 25%.

A.7 Experiment customization

The existing templates can be evaluated on any x86-64 user-space binary. The easiest way to evaluate the existing templates on a new binary is to use the *run_arbiter.py* helper script and specifying the VD template to use as well as a path to the new binary as argument.

The process of implementing a new VD template for a different CWE type has been described in the paper and demonstrated in the comments inside the *examples* directory. This process can be followed in order to evaluate ARBITER using a new template.

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

Spoki is a real-time reactive network telescope. It is written in C++ and based on the actor model to achieve high scalability. The artifacts also include Python tools to analyze Spoki log files, identify downloaders distributed by attackers, and fetch files referenced by the downloaders.

We used Spoki to collect the data for our paper over the course of three months. The artifact contains the source code. It can be used to collect the same information (given a suitable setup) and get started with the evaluation.

A.2 Artifact check-list (meta-information)

Compilation: C++-17 compiler, Python 3.

Run-time environment Linux. Capabilities to capture network traffic (telescope, root access).

Hardware Depends on your traffic volume. Processing traffic from a /24 should work with 4 cores of common server hardware.

Output Spoki writes observed events to log files, which can be analyzed by our Python tools.

Experiments We collected data over three months. The data sets we used contain sensitive data and are thus not available.

Publicly available? Yes, see below.

Code licenses MIT License

A.3 Description

A.3.1 How to access

All artifacts and future results are available via `https://spoki.secnow.net/`. We also archived the artifact on Zenodo: `https://zenodo.org/record/5702603`.

A.3.2 Software dependencies

Spoki runs on Linux and uses the following open-source software. The repository contains a script to build them.

- The C++ Actor Framework (CAF)
- Scamper
- libtrace

The Python tools depend on Kafka. Python-related dependencies can be installed via `pip` (see below).

A.4 Installation

Spoki The source code is located in the `spoki/` folder. On Ubuntu 20.04, you first need to install the following dependencies:

```
$ sudo apt install gcc g++ cmake git curl make
    libtool-bin automake libpcap0.8-dev
    libbison-dev flex
```

Now you can build the required libraries via a script in the repository. It installs the libraries into a local folder.

```
$ ./setup.sh
```

Finally, you need to configure and build Spoki:

```
$ ./configure
$ make -C build
```

The Spoki binary will be located at `./build/tools/spoki/`.

Evaluation Tools to download the malware linked in payloads are located in `evaluation`. First, setup a local virtual environment. Inside the environment you can setup the tools via the makefile (run it twice the first time):

```
$ make update
$ make update
```

This will link the following tools into the virtual environment:

assemble Reads Spoki logs and assembles events.

filter Identifies events with downloaders.

clean Extracts and clean links from events.

download Follows links and downloads executables.

Additional tools to analyze port statistics, contact types, query malware hashes in VirusTotal, and annotate data are located in the same project. Please check the `README.md` file for details.

A.5 Experiment workflow

Spoki can be run directly from the command line and accepts configuration via a `caf-application.conf` file. Please check the `README` of Spoki for the configuration details. It is necessary to configure the data source (e.g., the interface to read packets from), a folder for the logs, and a tag for your datasource. Spoki writes two types of logs: event logs that contain information on observed events alongside Spoki's probe reply and scamper logs that list the probe confirmations received from scamper.

The malware processing tools require a running Kafka instance for communication. They further accept CLI options to configure the Kafka topic they use between them. The `assemble` further accepts the location of the Spoki tools and a date and hour to start processing. These tools build a processing pipeline. In the final step, Spoki logs all observed URLs and the data it downloads alongside some meta information.

A.6 Evaluation and expected results

We cannot provide our data sets because they include personally identifiable information such as IP addresses. We provide Spoki's code to allow others to repeat our experiments.



A Artifact Appendix

A.1 Abstract

In this paper, we present the first techniques to automate the discovery of new censorship evasion techniques purely in the application layer. We present a general solution and apply it specifically to HTTP and DNS censorship in China, India, and Kazakhstan. Our automated techniques discovered a total of 77 unique evasion strategies for HTTP and 9 for DNS, all of which require only application-layer modifications, making them easier to incorporate into apps and deploy. We analyze these strategies and shed new light into the inner workings of the censors. We find that the success of application-layer strategies can depend heavily on the type and version of the destination server. Surprisingly, a large class of our evasion strategies exploit instances in which censors are more RFC-compliant than popular application servers.

For the purposes of this submission, our artifacts are (1) the strategies we present in the paper and (2) the code used to implement them. We developed our fuzzer by building off of the open-source Geneva project (`https://github.com/Kkevsterrr/geneva`), but our code has not yet merged into that repository publicly. Therefore, we have provided the full modified codebase to assist in the evaluation.

For this artifact evaluation, we demonstrate how the reader can evaluate (1) that our strategies can generate modified requests; (2) that our strategies can evade censorship. Optionally, the evaluator can test for themselves that our tool can fuzz HTTP requests.

A.2 Artifact check-list (meta-information)

- **Algorithm: A new algorithm for bypassing censorship with modifications to application-layer data.**
- **Program: The program—an extension to our prior work, Geneva—that implements the algorithm. Specifically, we are asking you to evaluate the engine that runs the strategies that our algorithm discovered, not to run the genetic algorithm that *found* the strategies.**
- **Security, privacy, and ethical concerns:**
- **Metrics: Whether or not it is able to access otherwise restricted content.**
- **Output: HTTP output (from running `curl`).**
- **Experiments: Re-run several of our censorship-evading strategies.**
- **How much disk space required (approximately)?: Very little (megabytes); a free-tier Ubuntu AWS instance would suffice.**
- **How much time is needed to prepare workflow (approximately)?: Minutes (set up an Ubuntu VM, install, and run).**
- **How much time is needed to complete experiments (approximately)?: Minutes, including setup.**
- **Publicly available (explicitly provide evolving version reference)?:** Yes
- **Code licenses (if publicly available)?:** BSD 3-Clause "New" or "Revised" License
- **Archived (explicitly provide DOI or stable reference)?:** `https://zenodo.org/record/6692160`

A.3 Description

A.3.1 How to access

The code is available at `https://zenodo.org/record/6692160`

We developed our fuzzer by building off of the open-source Geneva project (https://github.com/Kkevsterrr/geneva), but our code has not yet merged into that repository publicly. Therefore, we have provided the full modified codebase to assist in the evaluation.

Documentation for Geneva is available at `https://geneva.readthedocs.io/en/latest/`. To ease the burden on the artifact evaluators, we have provided just the required steps to evaluate our artifact below.

A.3.2 Software dependencies

See the dependency installation in the installation setup below (some basic python libraries).

A.3.3 Security, privacy, and ethical concerns

As this is accessing a server that we control, we do not anticipate any security, privacy, or ethical concerns. However, we do suggest that the evaluator run from a machine inside of a country that does not prosecute censorship circumvention (e.g., from within the United States).

A.4 Installation

1. **Setup a machine:** To test our artifacts, we recommend using Ubuntu 18.04, and although it may be possible to reproduce our results using a virtual machine, it is ideal if the machine has a public IP address and is not behind a NAT, to avoid any potential interference from your host or home network.

   We recommend setting up a free-tier Amazon EC2 machine with Ubuntu 18.04. Once your machine is up and running, transfer our artifact submission to it.

2. **Install dependencies:** Next, install the dependencies for Geneva:

   ```
   # cd geneva/
   # sudo apt-get install build-essential python-dev
   libnetfilter-queue-dev libffi-dev libssl-dev
   iptables python3-pip
   ...
   # python3 -m pip install -r requirements.txt
   ...
   ```

3. **Test strategies:** To validate our strategies, reviewers can test (1) that our code generates the strategies it says it does and (2) that these strategies are actually effective at evading censorship.

In our paper, we present over 85 strategies, and tested these across 8 servers against 3 different censoring regimes, using vantage points we obtained in those locations. Unfortunately, for security reasons, we cannot give evaluators direct access to these vantage points.

To make evaluation easier, we have set up an HTTP server running Apache 2.4.6 in Kazakhstan with `www.youporn.com` as the required Host header (we have provided the IP address and port of that server in our submission). Kazakhstan operates censorship bidirectionally (forbidden requests sent into the country are censored in the same way as requests leaving the country), which enables evaluators to trigger HTTP censorship remotely to our vantage point. If evaluators wish to test the rest of our strategies to the other censored countries or our DNS strategies, we can offer advice and work with the evaluators as to how to best purchase vantage points in those locations and test the remaining strategies.

With the server we set up, an evaluator can safely test a sample of our strategies to this IP address and verify that they do evade censorship. To maintain reviewer anonymity, we will discard this server's logs. In the below guide, we have removed our server's IP address; wherever you see `<ip>`, please replace with the IP address we provided in HotCrp

First, we will verify that you can reach our server. Start by `curl`ing to our server. Note that Since `curl` will not set a `Host` header by default, you should see a 403 Forbidden response:

```
$ curl <ip>:8000
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access /
on this server.</p>
</body></html>
```

Second, we will verify that you can experience censorship to our vantage point. Start `tcpdump` to our IP address in the background, and make a request with `www.youporn.com` in the `Host` header. You should see that censorship occurs (the censor null-routes our request), and the request is retransmitted:

```
$ sudo tcpdump -f -n 'host <ip>' &
$ curl -H "Host: www.youporn.com" <ip>:8000
03:31:11.724898 IP 172.172.172.172.38520 > <ip>.8000: Flags [S], seq
2244093827, win 62727, options [mss 8961,sackOK,TS val 585883751 ecr
0,nop,wscale 7], length 0
03:31:11.840224 IP <ip>.8000 > 172.172.172.172.38520: Flags [S.], seq
1501790452, ack 2244093828, win 65160, options [mss 1460,sackOK,TS val
2426661480 ecr 585883751,nop,wscale 7], length 0
03:31:11.840266 IP 172.172.172.172.38520 > <ip>.8000: Flags [.], ack 1,
win 491, options [nop,nop,TS val 585883867 ecr 2426661480], length 0
03:31:11.840609 IP 172.172.172.172.38520 > <ip>.8000: Flags [P.], seq
1:80, ack 1, win 491, options [nop,nop,TS val 585883867 ecr
2426661480],
length 79
03:31:12.219434 IP 172.172.172.172.38520 > <ip>.8000: Flags [P.], seq
1:80, ack 1, win 491, options [nop,nop,TS val 585884246 ecr
2426661480],
length 79
03:31:12.571446 IP 172.172.172.172.38520 > <ip>.8000: Flags [P.], seq
1:80, ack 1, win 491, options [nop,nop,TS val 585884598 ecr
2426661480],
length 79
03:31:13.275434 IP 172.172.172.172.38520 > <ip>.8000: Flags [P.], seq
1:80, ack 1, win 491, options [nop,nop,TS val 585885302 ecr
2426661480],
length 79
```

Shut down the `tcpdump` before continuing.

A.5 Experiment workflow

Next, we can evaluate that Geneva can implement the strategies in our paper. Since our paper reported over 85 strategies, to reduce the burden on the evaluators, we have sampled 3 strategies that work with this server and evade censorship in Kazakhstan.

- Strategy 1 (Long Request):

  ```
  [HTTPRequest:host:*]-insert{%20:start:value:1600}-|
  ```

- Strategy 2 (Host Header Whitespace):

  ```
  [HTTPRequest:host:*]-insert{%20:end:value:1}-|
  ```

- Strategy 3 (Request Line Whitespace):

  ```
  [HTTPRequest:method:*]- insert{%0A:start:value:1}-| \/
  ```

To use Geneva to test a strategy, we use Geneva's `--test-type` flag to invoke our HTTP fitness function (called `application_http`), and can test strategies with the following:

```
$ python3 evolve.py --test-type application_http --log debug --port 8000
--server <ip> --headers Host:www.youporn.com --eval-only "<STRATEGY HERE>"
```

Explanation of flags:

- `--test-type`: The fitness function plugin to invoke, in this case, `application_http`
- `--log`: the log level to use (debug, so the evaluator can see the strategy running and the exact request it sends on the wire)
- `--port`: the port the server is listening on (8000 for our server)
- `--server`: the server to test with; supply our IP address.
- `--eval-only`: this is a Geneva flag instructing it to evaluate a single strategy and then exit. Provide the strategy to test here.

A.6 Evaluation and expected results

Our primary claim is that these modifications to application-layer data permit evasion of censorship. You can use the above steps to run individual strategies (either the ones we selected, or any of the ones from the paper: you should be able to simply copy-paste them where it says "`STRATEGY HERE`") and should see something to the effect of the following:

```
$ sudo python3 evolve.py --test-type application_http --log debug
--port 8000 --server <ip> --headers Host:www.youporn.com --eval-only
"[HTTPRequest:host:*]-insert{%20:end:value:1}-|"
2022-06-22 20:51:13 DEBUG:Launching strategy evolution: --test-type
application_http --log debug --port 8000 --server <ip> --headers
Host:www.youporn.com --eval-only [HTTPRequest:host:*]-
insert{%20:end:value:1}-|
2022-06-22 20:51:13 INFO:Logging results to trials/2022-06-
22_20:51:13/logs
2022-06-22 20:51:14 DEBUG:Beginning evaluation in plugin
2022-06-22 20:51:14 DEBUG:Now entered evaluate of
ApplicationHTTPPlugin.
2022-06-22 20:51:14 DEBUG:Only using port 8000
2022-06-22 20:51:14 DEBUG:Starting with request: b'GET /
HTTP/1.1\r\nHost:www.youporn.com\r\n\r\n'
2022-06-22 20:51:14
DEBUG:-------------------------------------------
2022-06-22 20:51:14 DEBUG:Running individual:
[HTTPRequest:host:]-
insert{%20:end:value:1}-| \/
2022-06-22 20:51:14 DEBUG: + out action tree triggered:
[HTTPRequest:host:]-insert{%20:end:value:1}-|
2022-06-22 20:51:14 DEBUG:Inserting value: |%20| into
the end of the
variable header_value, 1 times, in the header
Host:www.youporn.com
2022-06-22 20:51:14 DEBUG:Shuffling headers...
2022-06-22 20:51:14 DEBUG:New request string: b'GET /
HTTP/1.1\r\nHost:www.youporn.com \r\n\r\n'
2022-06-22 20:51:14 DEBUG:Connecting to url <ip> with
port 8000
2022-06-22 20:51:14 DEBUG:Response data
HTTP/1.1 200 OK
Date: Thu, 23 Jun 2022 03:51:14 GMT
Server: Apache/2.4.6 (Unix)
Last-Modified: Thu, 23 Jun 2022 03:12:30 GMT
ETag: "2d-5e214d2fe4577"
Accept-Ranges: bytes
Content-Length: 45
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>

2022-06-22 20:51:14
DEBUG:===========================================
2022-06-22 20:51:14 DEBUG:EVADED the censor! Had
response line: HTTP/1.1
200 OK
2022-06-22 20:51:14
DEBUG:===========================================
2022-06-22 20:51:14 DEBUG:Punishing for complexity: 1
2022-06-22 20:51:14 DEBUG:New request is 1 bytes longer
than original.
Punishing -10 fitness.
2022-06-22 20:51:14 DEBUG:Individual
[HTTPRequest:host:]-
insert{%20:end:value:1}-| \/ ran with a fitness of: 316
2022-06-22 20:51:14 INFO:[316]
2022-06-22 20:51:14 INFO:Trial 0: success! (fitness
= 316)
2022-06-22 20:51:14 INFO:Overall 1/1 = 100%
2022-06-22 20:51:14 INFO:Exiting eval-only.
```

In the above output, you can see that the strategy was implemented on the outgoing request: one space (`%20`) was added to the outbound request at the end of the value of the `Host` header (`'GET / HTTP/1.1\r\nHost:www.youporn.com \r\n\r\n'`). You may use `tcpdump` to verify that these bytes are transmitted on the wire.

You can next see that this strategy evaded censorship, and the server was able to properly respond. You can also see that the fitness function correctly detected that it evaded censorship, and awarded a positive fitness value accordingly. Repeat this step for the three strategies we provided, and you can verify that each correctly implements the strategy it purports to on the wire and that those strategies successfully evade censorship.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact is separated into two parts: **simulations** (Sections 6 and 7 on the paper) and **evaluations** (Sections 8 and 9).

The simulations part assumes a machine with Python3.8 and the libraries *scipy*, *numpy*, *lightning-utils*, *networkx*, *matplotlib*, and *seaborn*. This part runs standalone simulations and was used to generate Figures 4, 5, 6, 7, and 10 on the paper. In practice, to reduce running time, we ran most of the simulations using Slurm on an internal cluster.

On the part of the evaluation, we implemented the Twilight system in 4 components: smart contract (Solidity), enclave (C++, SGX), relay (Python), and the evaluations manager (Python). The enclave and relay parts were run on an Azure VM. The tests were written using *pytest*, used Ganache as a local blockchain, and the compiler *solcjs* to compile the contract. In order to create fully reproduceable results, this evaluation part can be executed only in Azure cloud environment. The manager creates the relevant resources in the cloud and executes the system according to the evaluation experiments. To run this part, we assume a machine with Python3.8, the libraries *paramiko* (to establish SSH connection to the machines) and *matplotlib*, *seaborn* to generate the plots, the Pythonic Azure SDK, credentials with admin permissions (to create resources such as VMs, NICs, IPs, etc.), and enough quota in Azure to create these machines. To create Figures 8 and 9 in the paper, we used 6 VMs with the type *Standard_DC1s_v2* (3 in each region: *eastus* and *northeurope*).

A.2 Artifact check-list (meta-information)

- **Algorithm:** We implemented Algorithm 1 from the paper (Appendix, Section A) inside the SGX (file *Enclave/tree.cpp*), and Algorithm 2 (Appendix, Section B) inside the smart contract (file *smart_contract/channel.sol*).
- **Compilation:** Most of our code is written in Python. The part of the enclave is written in C++17 and could be compiled by running *make SGX_MODE=HW SGX_PRERELEASE=1* in the root directory. The smart contract can be compiled using any solidity compiler, we used *solcjs* without any flags.
- **Hardware:** The evaluation uses an Azure VM of type *Standard_DC1s_v2*, with SGX-1. The local code runs in Python and has no hardware requirements.
- **Experiments:** In the evaluation part, we created a line topology of 6 machines: Alice, 4 relays, and Bob. For every throughput value that has been tested, we executed a command on Bob's machine to initiate the given amount of payments every second. Then, after 10 seconds, we queried Alice's machine for the number of finished payments (to get the throughput) and for the duration of the payment (to get the latency).
- **How much disk space required (approximately)?:** Negligible. Less than 1GB should be sufficient.
- **How much time is needed to prepare workflow (approximately)?:** Creating and preparing all the cloud resources is being executed once, and take less than an hour overall.
- **How much time is needed to complete experiments (approximately)?:** Each data point in Figures 9 and 10 should be executed separately and takes around 20 minutes (can be controlled by lowering the number of repetitions. We used a default of 20 repetitions).
- **Publicly available (explicitly provide evolving version reference)?:** The Github repository is: https://github.com/saart/Twilight.
- **Code licenses (if publicly available)?:** None.
- **Data licenses (if publicly available)?:** We used the Lightning Network topology which was queried (using a standard CLI command https://github.com/lightningnetwork/lnd/blob/593962b788753768661582d11221f32ebf7dbe67/cmd/lncli/commands.go#L1515) from a Lightning node. This is publicly available.
- **Workflow frameworks used?:** We used Python and Azure's SDK to manage the experiments (initiate and teardown machines), FastAPI (https://fastapi.tiangolo.com/) as the communication framework between the relays, and Pistache (https://github.com/pistacheio/pistache) as the communication framework between the relay and the enclave.
- **Archived (explicitly provide DOI or stable reference)?:** On the paper we used tag: https://github.com/saart/Twilight/tree/USENIX-Security-22.

A.3 Description

A.3.1 How to access

Publicly available at: https://github.com/saart/Twilight

A.3.2 Hardware dependencies

None (run on machines with specific requirements on the cloud).

A.3.3 Software dependencies

Python3.8 with the libraries *scipy*, *numpy*, *networkx*, *matplotlib*, and *seaborn*. For the evaluation part, Azure SDK is also required. Moreover, we assume network connectivity, and in particular the ability to run Azure CLI command and establish SSH sessions to Azure's VMs.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

To install the Azure VM follow the description in: https://github.com/saart/Twilight#how-to-install-on-a-new-azure-confidetial-computing-machine. To install the local machine: Install Python3.8 with https://www.python.org/downloads/, install setuptools using *pip install setuptools==58.0.0*, install azure cli using *pip install azure-cli==2.19.0* and install the rest of requirements using *pip install -r simulations/requirements.txt*. Then, create permissions from your Azure profile using https://docs.microsoft.com/en-us/azure/developer/python/sdk/authentication-overview (make sure that you can authenticate using python, e.g. by running the Pythonic command *get_client_from_cli_profile(ComputeManagementClient)*).

A.5 Experiment workflow

The simulation workflow is standalone per Figure:

- Figure 4 can be reproduced using *simulations/distinct_routes.py*
- Figures 5 and 6 can be reproduced using *simulations/visualizations/efficiency_privacy_tradeoff.py*
- Figure 7 can be reproduced using *simulations/noise_simulations/liquidity_distribution.py* and then *simulations/noise_simulations/success_rate.py*
- Figure 10 can be reproduced using *simulations/visualization/adoption.py*

The evaluation workflow is managed using the script *simulations/manage_tests.py*.The general flow is:

1. Create the machines in the relevant regions (*eastus* and *northeurope*).
2. Start/Restart the machines, which starts two processes: the relay and the enclave as system services. This step also refreshes the existing machines between different experiments.
3. Build the P2P and channels topology. I.e. query for the names of the relays, and execute *register* to create the edges.
4. Execute a command on Bob's machine, that initiates a repetitive thread that starts payments according to the given route and the given desired throughput.
5. Query Alice's machine on the payments that have been finished in the last few seconds, and store the throughput (number of finished payments) and the latency (duration of each payment).

Steps 2-5 are re-executed repetitively: both to evaluate the same experiment again, and to evaluate using different parameters (number of issued payments, route length).

In order to plot the results and reproduce Figures 8 and 9, use the file *simulations/draw_evaluation_figures.py*.

A.6 Evaluation and expected results

Our main claim in the paper is that Twilight is a valid solution to the probing attack of off-chain networks.

This claim is backed by this artifact that presents both simulations and evaluations of the system and its properties. Every one of the Figures 3-10 from the paper is backed with the code that is presented in this artifact.

To reproduce the results of the simulations part (Figures 3-7 and 10), follow the description in Section A.5, and run each file to generate the corresponding figure.

To reproduce the results of the evaluation part (Figures 8 and 9), first connect to your azure environment using the bash command *az login*, authenticate in the opened browser, and run the evaluation using the script *simulations/manage_tests.py* from the directory *simulations/*. Then, draw the plots using *draw_evaluation_figures.py*.

The results from both parts should plot the same graphs that we presented on the paper. This is possible that the results will vary, therefore for each plot on the paper we included error bars that should present the range of the variation.

A.7 Experiment customization

The topology of the evaluation is flexible. Although we presented on the paper only a line-topology, we included in the file *simulations/manage_tests.py* more possible topologies that evaluate different use-cases. The different use cases that we also examined are: changes in the throughput over time, building a topology based on the topology of the Lightning network, and two routes that intersect in the middle (X topology).

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

In this work, we developed a voting-based domain ranking method that operates on passive DNS (PDNS) data to construct a domain top list. We open-source our top list construction implementation at *https://github.com/SecRank/secrank-sourcecode/releases/tag/v1.0.0*, to provide transparency into the design of our ranking method. The code provided (written in Scala) processes proprietary PDNS data to compute a daily top 1M domains list, running in a distributed fashion using Apache Spark on YARN. As we are unable to release the raw proprietary PDNS data used, the code is not directly runnable. Instead, it serves as a reference for understanding the details of our ranking method, and as a template that can be modified for other PDNS datasets and computing environments.

As our top list design achieves favorable stability and manipulation resistance properties, we also provide public access to a regularly updated domain top list constructed using our ranking method at *https://secrank.cn/topdomain*, for others to use in their research. After registering for an account on the website, a user can download a daily top 1M domains list as well as historical top lists.

A.2 Artifact check-list (meta-information)

- **Algorithm:** We provide a new voting-based domain ranking algorithm that operates on PDNS data, where the domain preferences of individual IP addresses are first computed, and then a global top list ranking is produced by applying a voting scheme across all IP addresses.
- **Security, privacy, and ethical concerns:** This artifact does not raise any security, privacy, or ethical concerns.
- **Publicly available?:** Our top list construction implementation is publicly available at https://github.com/SecRank/secrank-sourcecode/releases/tag/v1.0.0. A regularly updated domain top list constructed using our ranking method is publicly available at *https://secrank.cn/topdomain*, upon registering for a user account.
- **Code licenses?:** MIT License.
- **Archived?:** Our top list construction implementation is publicly archived at https://github.com/SecRank/secrank-sourcecode/releases/tag/v1.0.0.

A.3 Description

A.3.1 How to access

Source Code Access. We open-source our top list construction implementation at *https://github.com/SecRank/secrank-sourcecode/releases/tag/v1.0.0*. Our implementation relies on proprietary PDNS data, which we are unable to release for privacy and commercial reasons. Thus, the code is not directly runnable, and rather provides transparency into the design of our domain ranking method.

Those interested may adapt our code for their own PDNS data/format and computing environment. Note that our implementation uses Apache Spark on YARN, with input and output data stored in HDFS. For users with a similar computing environment, our code can be most directly applied by providing the proper input and output data file paths, as well as adjusting the data field names extracted from the input data. Further details are provided in the *README.MD* file.

The code repository consists of the following main files:

- *README.MD*: The README file providing guidance on using the code.
- *TopFQDNDailyRelease.scala*: The main algorithm source code file, containing detailed comments for each algorithm component that reference the relevant sections in our paper describing the algorithm's design. We additionally document the input/output file paths and data formats that must be modified if adapting this code for other PDNS datasets.
- *pom.xml*: The XML file that contains information about the software package and configuration details (including software and library dependencies).
- *submit.sh*: The shell script to submit the Spark application to a YARN cluster.

Daily Top 1M Domains List Access. Public access to a regularly updated top 1M domains list is available at https://secrank.cn/topdomain, through registering for a free account. With an activated account, users can download the latest daily list as well as historical lists.

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

Our released implementation is written in Scala and runs on Apache Spark on YARN. While our released code is not directly runnable, those modifying it for use will likely require IntelliJ IDEA, Java 1.8, Maven JDK 1.8, Scala 2.11.8, Apache Spark 2.4.5, and Hadoop 2.7.2. (These dependencies are also shown in the *pom.xml* file in the release package.)

A.3.4 Data sets

The source code relies on proprietary PDNS data, which we are unable to release for privacy and commercial reasons.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

While our released code is not directly runnable (as we cannot release our raw input PDNS data), we provide guidance on how one could modify the code to run on their own input PDNS dataset. As our implementation is executed via Apache Spark on YARN, we assume a similar computing environment (i.e., Java 1.8, Maven JDK 1.8, Scala 2.11.8, Apache Spark 2.4.5, and Hadoop 2.7.2).

1. We suggest using IntelliJ IDEA to create an Apache Maven project, and replacing the default *pom.xml* file with the *pom.xml* file in our Github repository (which contains all dependency configurations and package requirements).
2. Next, place *TopFQDNDailyRelease.scala* in the path *$PROJECT_PATH$/src/main/java/com/secrank/examples/*.
3. In *TopFQDNDailyRelease.scala*, modify the *trends_path* and *access_path* variables to reference the input PDNS data file paths on HDFS, and also modify accordingly the output file path (in the code's final stage).
4. As documented in the comments of *TopFQDNDailyRelease.scala*, the code assumes certain fields are present in the input data format. If those fields are not available, either the source data must be modified to provide these fields, or the field names must be adjusted accordingly in the code to reflect the source data.
5. After modifying the code, package the Maven project into a JAR file, upload this JAR file to your Spark client, and execute *submit.sh* to submit the Spark application to the YARN cluster. After the code fully executes, the output will contain the top 1M domains list.

A.5 Experiment workflow

N/A

A.6 Evaluation and expected results

Users are expected to modify the code for their own input PDNS data and computing environments, with the expected output being a top 1M domains list computed using our domain ranking method.

A.7 Experiment customization

N/A

A.8 Notes

N/A

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact contains the source code of all the experiments that were used to produce the figures and numbers in the research paper. Its hardware requirements are a bare-metal machine with an Intel i7-9700 CPU (however, with some engineering efforts, the code can be ported to other Intel CPU models), and a second machine that can communicate with (i.e., send network requests to) the first machine. Its software requirement is Ubuntu 18.04 or 20.04 with its default system configuration. It will take approximately 10 days to reproduce the entire set of all the experiments in the paper.

A.2 Artifact check-list (meta-information)

- **Compilation:** GCC and Golang.
- **Hardware:** Two machines:
  1. A bare-metal machine with Intel i7-9700. However, with additional engineering efforts, the code can be ported to other CPU models (e.g., see Table 1 in the paper).
  2. A second machine that can communicate with (i.e., send network requests to) the first machine.
- **Security, privacy, and ethical concerns:** In the proof-of-concept attacks, you will launch both the attacker and the victim. No production servers are targeted.
- **Experiments:** Please checkout the README in the artifact for details on the experiments included.
- **How much disk space required (approximately)?:** We recommend at least 2 GB of free disk space.
- **How much time is needed to prepare workflow (approximately)?:** Overall, preparing the various workflows should take approximately 30 minutes on the i7-9700 CPU.
- **How much time is needed to complete experiments (approximately)?:** Reproducing all the experiments will take approximately 10 days.
- **Publicly available (explicitly provide evolving version reference)?:** Yes, the artifact is publicly available at `https://github.com/FPSG-UIUC/hertzbleed`
- **Code licenses (if publicly available)?:** University of Illinois / NCSA Open Source License.
- **Archived (explicitly provide DOI or stable reference)?:** `https://github.com/FPSG-UIUC/hertzbleed/releases/tag/usenix2022ae`

A.3 Description

This artifact includes (i) several experiments that reverse engineer the dependency between data, power and frequency on Intel CPUs and (ii) proof-of-concept attacks that leak full cryptographic keys from two SIKE libraries, break KASLR, and establish a covert channel using the frequency side channel.

A.3.1 How to access

`https://github.com/FPSG-UIUC/hertzbleed`.

A.3.2 Hardware dependencies

1. A bare-metal machine with Intel i7-9700. However, with additional engineering efforts, the code can be ported to other CPU models (e.g., see Table 1 in the paper).
2. A second machine that can communicate with (i.e., send network requests to) the first machine.

A.3.3 Software dependencies

A default installation of Ubuntu 18.04 or 20.04, and (if not pre-installed) the programs `gcc`, `golang`, `stress-ng`, and `python3`.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

No production servers are targeted. No sensitive data is collected, and no damage is caused to the machines.

A.4 Installation

Please checkout the README in the artifact for instructions on how to install the software dependencies.

A.5 Experiment workflow

Please checkout the README in the artifact for instructions on how to set up the experiment workflow.

A.6 Evaluation and expected results

The expected results are figures like the ones in the research paper and numbers like the ones reported in the research paper. Some variability is possible due to environmental/machine differences, but the general figure trends should apply.

A.7 Experiment customization

Please checkout the README in the artifact for instructions on how to customize the experiments.

A.8 Notes

N/A

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

The artefact consists of scripts for collecting several datasets of live-defended VPN network traces using the QCSD framework, simulating defended network traces, and performing machine-learning evaluations, in addition to the source code of the QCSD client library and test clients written in Rust and the datasets collected during the evaluation. The artefact requires at least 2 CPU cores and 4 GB of memory, however additional cores help greatly to reduce run times, as does access to GPUs. It requires python3.8, rust, and docker and was tested on Ubuntu 20.04. The artefact generates the plots present in the paper and allows running the machine-learning evaluations on the datasets from the paper to compare the resulting plots to those in the paper.

A.2 Artifact check-list (meta-information)

- **Compilation: rustc >= 1.51, publicly available**
- **Data set: included**
- **Run-time environment: root access, Ubuntu 20.04, docker, rust, python3.8, git/git-lfs**
- **Hardware: Wireguard VPN servers, GPU**
- **Run-time state: impacted by network throughput**
- **Execution: test → under 24 hours; full → data collection > 4 days with 3 VPN gateways and a 32 core server, machine-learning evaluations several days on an RTX 3060**
- **Security, privacy, and ethical concerns: network scanning, web-page crawling**
- **Metrics: recall, *r*-precision, Pearson's correlation, LCSS**
- **Output: plots, tables, see paper for expected results**
- **Experiments: automated setup, snakemake workflow**
- **How much disk space required (approximately)?: 60 GiB**
- **How much time is needed to prepare workflow (approximately)?: 1 hour**
- **How much time is needed to complete experiments (approximately)?: test → under 24 hours, full → upwards of 6 days**
- **Publicly available (explicitly provide evolving version reference)?: yes, `https://github.com/jpcsmith/qcsd-experiments/tree/v1.0.1`**
- **Code licenses (if publicly available)?: MIT, Apache-2.0**
- **Data licenses (if publicly available)?: Creative Commons Attribution 4.0 International**
- **Workflow frameworks used?: Snakemake**
- **Archived (explicitly provide DOI or stable reference)?: `10.3929/ethz-b-000565356`**

A.3 Description

A.3.1 How to access

The repository containing the scripts can be by cloned from the GitHub repository `https://github.com/jpcsmith/qcsd-experiments.git`, with tag v1.0.1 corresponding to the version of this appendix. The dataset collected during the paper, along with archived versions of the associated repositories and a virtual machine with the software dependencies installed, can be found under the DOI `10.3929/ethz-b-000565356`.

A.3.2 Hardware dependencies

Below we describe the hardware dependencies based on the various phases in the workflow.

Dataset collection The provided test configuration runs on a server with 8 CPU cores and 8 GB of memory, and with a single Wireguard VPN gateway running on the same host. The full collection utilised 3 VPN gateways (2 CPU cores and 2 GB memory is more than sufficient for each) and 12 VPN clients per gateway (36 in total) running on a server with 32 cores and 188 GB of memory. Each VPN client is is restricted to at most 2 CPU cores. Reduce the number of clients per gateway to use less cores at the cost of longer dataset collection times.

Machine learning (ML) evaluations The ML evaluations associated with the test configuration can be run on 8 cores or less. For the full configuration, at least 1 GPU is recommended such as an RTX 3060 or better.

A.3.3 Software dependencies

The following software dependencies are assumed to be already installed, and are installed in the provided VM:

- *Ubuntu 20.04 and bash*: All code was tested on a fresh installation of Ubuntu 20.04.
- *git, git-lfs*: Used to clone the code repository and install python packages.
- *Python 3.8 with virtual envs*: Used to create a Python 3.8 virtual environment to run the evaluation and collection scripts.
- *docker >= 20.10*: Used to isolate simultaneous runs of browsers and collection scripts, as well as to enable multiple Wireguard clients on a single host. The user must be able to manage containers without using sudo.
- *tcpdump >= 4.9.3*: Used to capture traffic traces. Must be configured to allow the non-root user to capture.
- *rust (rustc, cargo) == 1.51*: Used to compile the QCSD library and test client library written in Rust.
- *Others*: Additionally, the following packages are required to build the QCSD library and test client, and can be installed with the ubuntu package manager, apt: `build-essential mercurial gyp ninja-build libz-dev clang tshark texlive-xetex`

Other software dependencies, such as ansible and Wireguard, are installed automatically.

A.3.4 Data sets

The dataset from the collection performed in the paper has the DOI `10.3929/ethz-b-000565356` and can be downloaded and used as a starting point for running the evaluations. To do so, replace the `results/` directory in the cloned repository with the results directory found in the gzipped tar archive.

A.3.5 Security, privacy, and ethical concerns

The evaluation downloads thousands of web page HTMLs and associated resources. The scripts in the workflow avoid overloading servers by scheduling requests such that sequential requests to a domain are either delayed or interleaved with requests to different domains. Additionally, be aware of any regulations of your network provider regarding performing automated web browsing.

A.4 Installation

If using the provided VM image, change to the home directory of the vagrant user `/home/vagrant/`, otherwise change to the directory in which you would like to install the artefact.

1. Clone the repository `https://github.com/jpcsmith/qcsd-experiments.git` using `git clone`.
2. Change to the code directory and pull the additional resources `cd qcsd-experiments && git lfs pull`.
3. If you want to use a specific version of the repository, change to it now (e.g., `git checkout v1.0.1`).
4. Create a Python 3.8 virtual environment and activate it `python3.8 -m venv env && source env/bin/activate`
5. Upgrade the python package manager (`python -m pip install -U pip wheel`) and install required python packages `python -m pip install --no-cache-dir -r requirements.txt`.

Decide whether you want to run the experiments locally or distributed across multiple machines. The file `ansible/distributed` contains an example of the configuration required for running with remote VPN gateways and clients. The file `ansible/local` contains the configuration for running the experiments locally, and is used as an example for the following steps.

6. Set the `gateway_ip` variable in `ansible/local` to the non-loopback IP address of the host, for example, the LAN IP address.
7. Change the `exp_path` variable to a path on the (local) filesystem. It can be the same path to which the repository was cloned.
8. Run the command `ansible-playbook -i ansible/local ansible/setup.yml` to setup the docker image for creating the web-page graphs with Chromium; create, start, and test docker images for the Wireguard gateways and clients; and download and build the QCSD library and test clients.

The QCSD source code is cloned on the remote host into the third-party/ directory of the folder identified by the `exp_path` variable in the hosts file (`ansible/local` or `ansible/distributed`).

A.5 Experiment workflow

Before running the workflow, it is necessary to ensure that the appropriate environment variables are set. This can be done with `source env/bin/active` to activate the python environment created during installation, and `source env_vars` to set environment variables for the project.

The results and plots in the paper were produced using snakemake. Like GNU make, snakemake will run all dependent rules necessary to build the final target. The general syntax is

```
snakemake -j --configfile=<filename> <rulename>
```

where `<filename>` can be `config/test.yaml` or `config/final.yaml` and `<rulename>` is the name of one of the snakemake rules found in `workflow/rules/*.smk` files or the target filename. Table 1 lists the figures in the table and the rules to produce them, whereas the following section describes the results in the paper and the rule used to produce them. The listed output files can be found in the results directory.

Generally, the various result workflows can be divided into the phases: scan, collect, evaluate. In the first phase, scan, a python script is used to scan domains from the Alexa Top list for QUIC support. In the second phase, collect, Chromium browser instances are used to download the domains and record their resource dependencies, and then these dependency graphs are used to download live-defended and undefended samples using the QCSD test clients. Finally in the last phase, machine learning, overhead, or shaping evaluations are performed and plots are created.

A.6 Evaluation and expected results

The main claims and associated results are described below, along with the snakemake rules used to run them in parentheses. The snakemake rules can be tested with `snakemake -j --configfile=config/test.yaml <rulename>`, where `<rulename>` is given in parentheses below. The claims can be evaluated fully using `final.yaml` instead of `test.yaml`.

- *Claim*. QCSD can successfully emulate website-fingerprinting defences such as Tamaraw and FRONT.

  Results. The Pearson correlation coefficient indicate medium and strong correlations between simulated and live-defended traces with QCSD at 50 ms sampling rates. LCSS scores indicate long common sub-sequences at 5 ms sampling rates ($> 85\%$) (`shaping_eval__all`).

- *Claim*. QCSD adds small overheads to chaff-only defences such as FRONT.

  Results. Compared to the simulated FRONT defences, the live-defended traces increase bandwidth overhead by around 30% of the original trace length and latency overhead by under 10% (`overhead_eval__table` and `overhead_eval_mconn__table`).

- *Claim*. QCSD effectively defends single connections with FRONT, but only mildly reduces classification performance when defending with Tamaraw.

  Results. In the single connection setting, the r_{20}-precision-recall curves for traces defended with FRONT match or surpass the curves of the simulated FRONT defences, whereas for the Tamaraw defence, the curves curves of the live-defended

Table 1: List of figures and snakemake rules to produce them. Use `snakemake -j --configfile=<config> <rulename>` with the appropriate rule name to create the figure. Output file paths are relative to the `results/` directory.

Section	Figure	Rule name	Output file(s)
5. Shaping Case Studies: FRONT & Tamaraw	Figure 3	`shaping_eval__all`	`plots/shaping-eval-front.png`, `plots/shaping-eval-tamaraw.png`
	Table 2	`overhead_eval__table`	`tables/overhead-eval.tex`
6.1. Defending Single Connections	Figure 4	`ml_eval_conn__all`	`plots/ml-eval-conn-tamaraw.png`, `plots/ml-eval-conn-front.png`
6.2. Defending Full Web-Page Loads	Figure 5	`ml_eval_mconn__all`	`plots/ml-eval-mconn-tamaraw.png`, `plots/ml-eval-mconn-front.png`
	Figure 6	`ml_eval_brows__all`	`plots/ml-eval-brows-front.png`
E. Overhead in the Multi-connection Setting	Table 3	`overhead_eval_mconn__table`	`tables/overhead-eval-mconn.tex`
F. Server Compliance with Shaping	Figure 8	See `failure-analysis.ipynb`	`plots/failure-rate.png`

traces do not indicate significantly better performance than undefended traces (`ml_eval_conn__all`).

- *Claim*. QCSD effectively defends multiple connections with FRONT, but does not reduce classification performance when defending with Tamaraw.

  Results. In the single connection setting, the r_{20}-precision-recall curves for traces defended with FRONT match or surpass the curves of the simulated FRONT defences, both when considering multiple orchestrated connections based on dependency graphs (`ml_eval_mconn__all`) and in the browser setting (`ml_eval_brows__all`). When applying the Tamaraw defence on multiple orchestrated connections, the precision-recall curves are similar to the curves for the undefended traces for 2 of the 3 evaluated classifiers (k-FP and VarCNN) (`ml_eval_mconn__all`).

A.7 Experiment customization

The experiments can be customised by modifying the hosts on which the experiments are to be run (e.g., `ansible/local` and `ansible/distributed`) or changing experiment parameters in the snakemake config files (e.g., `config/test.yaml` and `config/final.yaml`).

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

In this artifact, we will build our Cheetah framework, and to evaluate three neural networks, i.e,. ResNet50, DenseNet121, and SqueezeNet in a secure two-party computation manner. Also, we build the counterpart (i.e., SCI-HE from the CrypTFlow2's paper) for comparison. Specifically, this artifact can reproduce the performance numbers in Table 8, and Fig 10 in our paper.

To build our programs, we require a C++ toolchain including `cmake (version>=3.13)`, C++ compiler that supports C++17 (e.g., `g++>=8.0`), `make` and `git`. Also we require `OpenSSL` to be installed. To achieve the best performance, or to reproduce the performance numbers in our paper, we expect the AVX512 instructions (i.e., avx512dq and avx512ifma) are enabled.

For each neural network, we will generate two executables, one for Cheetah and the other for CrypTFlow2's counterpart. Running the executable, it will log the running time and communication cost for evaluating the neural network securely. All the logs are re-directed to file.

A.2 Artifact check-list (meta-information)

- **Algorithm:**
  Our artifact includes all the proposed algorithms in our paper. Specially, the linear protocols (Fig2, Fig4, and Fig 11) are placed in `include/gemini/cheetah/` and the non-linear protocols (Fig8 and Fig 9) are placed in `SCI/src/Millionaire/` and `SCI/src/Nonlinear/`.
- **Compilation:**
  For compilation, we provide two scripts `scripts/build-deps.sh` and `scripts/build.sh` which builds the dependencies and our implementation, respectively.
- **Binary:**
  Using our scripts, the generated binaries are placed in the `build/bin/` directory, including 6 demo.
  - `sqnet-cheetah` Run inference on SqueezeNet using Cheetah.
  - `resnet50-cheetah` Run inference on ResNet50 using Cheetah.
  - `densenet121-cheetah` Run inference on DenseNet121 using Cheetah.
  - `sqnet-SCI_HE` Run inference on SqueezeNet using SCI-HE
  - `resnet50-SCI_HE` Run inference on SqueezeNet using SCI-HE.
  - `densenet121-SCI_HE` Run inference on SqueezeNet using SCI-HE.
- **Model:**
  We provide three pretrained neural networks:
  - `pretrained/sqnet_model_scale12.inp`,
  - `pretrained/resnet50_model_scale12.inp`,
  - `pretrained/densenet121_model_scale12.inp`.
- **Run-time environment:** >=2.70 GHz CPU with more than 16GB RAM. A Linux-like OS is preferred. For instance, our timing results can be reproduced using Alibaba Cloud ecs.c7.2xlarge instances or Amazon AWS c6g.2xlarge instances.
  If to execute our artifacts on a single machine (i.e., using two processes to mimic two remote machines), we recommend the CPU supports more than 8 cores.
- **Execution:**
  We provide two scripts `scripts/run-server.sh` and `scripts/run-client.sh` to execute our demo. For example, to run an inference on SqueezeNet using Cheetah. We can run `bash scripts/run-server.sh cheetah sqnet` on one terminal and run `bash scripts/run-client.sh cheetah sqnet` on other terminal.
  Replacing the first argument `cheetah` with `SCI_HE` to run CrypTFlow2's counterpart.
- **Metrics:**
  We measure the total running time and communication cost for one inference. The one-time setup including base-ot and key-generation are NOT included. Our programs will log the running time (in seconds) and communication (in megabytes) for each layer in the neural network.
- **Output:**
  Our programs will generate a detailed log for each layer including the running time and communication. Also, on the client side, it will output the prediction label for the input image. For example, after running the script `scripts/run-client.sh cheetah sqnet`, the generated log file `cheetah-sqnet_server.log` is placed under the current directory.
- **Experiments:**
  Our artifact reproduces some empirical results in our paper, including the Cheetah and SCI-HE performance numbers in Table 8, and the top-10 values in the final prediction vectors in Figure 10.
- **How much disk space required (approximately)?:**
  About 500 megabytes, including the source codes, dependencies, built objects and pretrained models.
- **How much time is needed to prepare workflow (approximately)?:**
  It took us less than 10 minutes to build all the programs. Note that to build our programs, we need to fetch dependencies from Github.
- **How much time is needed to complete experiments (approximately)?:**
  It might take about 15–30 minutes to run all the demos.
  The three Cheetah-related demos takes less than 4 minutes to execute on LAN and AVX512 enabled. While the three SCI-HE -related demos takes more than 10 minutes to execute on LAN and AVX512 enabled.
  If AVX512 is not available, the execution time might be twice.

- **Publicly available (explicitly provide evolving version reference)?:**

  Our source codes are available in https://github.com/Alibaba-Gemini-Lab/OpenCheetah, commit hash a9b362e.

A.3 Description

A.3.1 How to access

Our source codes are available in https://github.com/Alibaba-Gemini-Lab/OpenCheetah, commit hash a9b362e.

A.3.2 Hardware dependencies

To reproduce the performance numbers in our paper, we require the CPU to support AVX512, ie., avxdq and avx512ifma instructions. Nevertheless, our programs can run without AVX512 support.

A.3.3 Software dependencies

Our programs depend on the following open-sourced libraries. Note that we provide a script `script/build-deps.sh` to fetch and build these dependencies automatically.

- emp-tool https://github.com/emp-toolkit/emp-tool
- emp-ot https://github.com/emp-toolkit/emp-ot
- Eigen https://github.com/libigl/eigen
- SEAL https://github.com/microsoft/SEAL
- zstd https://github.com/facebook/zstd
- hexl https://github.com/intel/hexl/tree/1.2.2

A.3.4 Data sets

'N/A'

A.3.5 Models

We provide three pretrained neural networks:

- `pretrained/sqnet_model_scale12.inp`,
- `pretrained/resnet50_model_scale12.inp`,
- `pretrained/densenet121_model_scale12.inp`.

These pretrained model are generated/taken from CrypTFlow2's code base https://github.com/mpc-msri/EzPC/tree/master/Athos/Networks.

A.3.6 Security, privacy, and ethical concerns

'N/A'

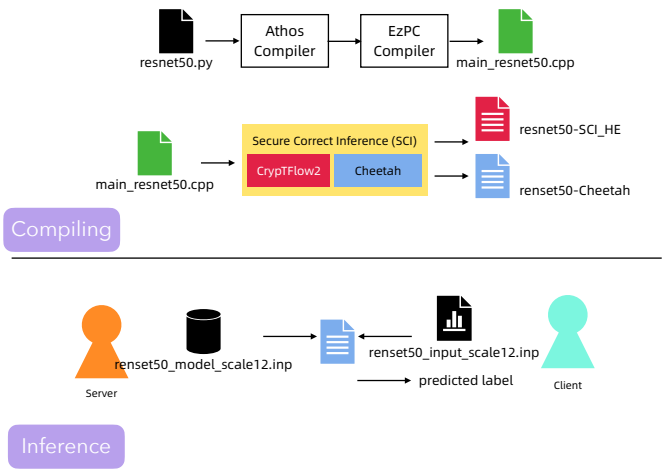


Figure 1: Workflow of Cheetah

A.4 Installation

1. Install the following requirements manuallly on your OS:
   (a) `git` We use `git` command to fetch all the source codes from Github.
   (b) `cmake` version >= 3.13. We use the `cmake` build-system to manage the source codes.
   (c) `make` To run the generated build scripts from `cmake`, we use the `make` command.
   (d) `bash` Helper scripts are written in `bash` syntax.
   (e) `openssl`. The openssl library should be installed in a 'standard' path (e.g., `/usr/include/`) so that `cmake` can find out where it is.
   (f) C++ compiler e.g. `g++` (on Linux) or `clang` (on MacOS). We require the C++ compiler to support at least C++-17. For example, `g++-8` and `clang-13`.
2. Fetch the Cheetah repo from Github via `git clone git@github.com:Alibaba-Gemini-Lab/OpenCheetah.git` Then go into the `OpenCheetah/` directory, and checkout `git checkout a9b362e` the specific version.
3. Build the dependencies via `bash scripts/build-deps.sh` This step will fetch many libraries from Github and build them, which might take a while to run.
4. Build the executables via `bash scripts/build.sh` This step will build 6 executables placed in the `build/bin/` directory.

A.5 Experiment workflow

From the high-level view, the current Cheetah implementation is an alternative implementation of the Secure and Correct Inference (SCI) Library [3]. We keep using the same interface of SCI so that we can leverage the Athos compiler [1] and the EzPC compiler [2] to convert a Python script that defines the structure of a neural network using TensorFlow to a secure two party computation C++ program that evaluates that neural work. A such compilation takes place once

for one neural network, and no trained model or input is needed during the compilation. In this artifact, we place the pre-compiled neural networks under the folder `networks/`.

For the secure inference, the server and the client run the compiled program with their private input. Also, input of server (i.e., pretrained model) and input of client (i.e., image) are also pre-processed using Athos's script. For example, floating point values are pre-processed to fixed-point values. The program will read the input from `stdin`. Also the program requires many parameters to run

- 'r' The role of player, 'r=1' indicates server and 'r=2' indicates client
- 'k' The fixed-point precision.
- 'ip' The IP address of the server.
- 'p' The port for the client's program to connect.
- 'nt' The number of threads. We can set at most 4 threads.
- 'ell' The bit length for the secret sharing, e.g,. we use 'ell=37' in our paper.

We provide helper scripts in `scripts/run-client.sh` and `scripts/run-server.sh` which hide most of the details for this parameters setting.

A.6 Evaluation and expected results

In our paper, we majorly claim two points.

1. Cheetah can evaluate deep neural network in minutes. For instance, in Table 8, we claim that Cheetah can evaluate ResNet50 within 1.5 minutes and transfer about 2.3 GB messages over LAN.
2. Our one-bit approximate truncation is effective for deep neural network inference. In § 6.5, we state that Cheetah can output almost the same prediction vector as SCI (which is bit-wise equivalent to the plaintext fixed-point computation).

By running our artifacts, we can reproduce the results in Table 8 and Figure 10.

To run our artifacts **locally**, execute as follows (take ResNet50 as the example)

1. Run `bash scripts/run-server.sh cheetah resnet50` on one terminal.
2. Run `bash scripts/run-client.sh cheetah resnet50` on the other terminal.

By replacing `cheetah` as `SCI_HE`, it will run the SCI-HE's counterpart. The other pretrained models, i.e., SqueezeNet (`sqnet`), DenseNet121 (`densenet121`) can be used by switching the second parameter.

After the computation is done, a log file is generated under the current directory, e.g., `cheetah-resnet50_client.log` on the client's machine and `cheetah-resnet50_server.log` on the server's machine. These files contain a detailed log for each layer of the neural network which can be used to validate the numbers in Table 8 and Figure 10 in our paper. The total computation time can be found in client's log file. For example in the 273-th line of `cheetah-resnet50_client.log`, it might record `Total time taken = 80719 milliseconds`. The total communication cost can be found in server's log file. For example in the 276-th line of `cheetah-resnet50_server.log`, it might record `Total comm (sent+received) = 2289.33 MiB`. The computation time might vary within 10% while the communication cost barely change much.

In addtion, we also print out the top-10 values in the final prediction vector to the last three lines in the client's log file. For our ResNet50 example, it will record

```
top-10 values from ResNet50
[13.0649084,11.7061750,10.7425666,10.4339929,9.8536843,···
predicted label=249
```

In the `SCI_HE-resnet50_client.log` (generated by running the `resnet50-SCI_HE` demo), it records

```
top-10 values from ResNet50
[13.0845959,11.7224159,10.7543676,10.4407995,9.8753999,···
predicted label=249
```

This reproduces the numbers in Figure 10 of our paper.

A.7 Experiment customization

If running on two remote machines, we first edit the `SEVER_IP` and `SERVER_PORT` variables defined in `scripts/common.sh`. The `scripts/throttle.sh` script can be used to manipulate the bandwidth (i.e., speed and ping latency). We can used this script to mimic the WAN/LAN setting within lab enviorments, e.g., running program within one machine. For example, using

```
sudo scripts/throttle.sh wan
```

on a Linux OS which will limit the local-loop interface to about 400Mbps bandwidth and 40ms ping latency. You can check the ping latency by just ping 127.0.0.1. The bandwidth can be check using extra `iperf` command.

A.8 Notes

To reproduce the timing numbers in our paper, we require the AVX512 instructions (i.e., avx512dq and avx512ifma) are supported. If AVX512 is not available, the timing numbers (both for Cheetah and SCI-HE) will be increased about 2×.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

References

[1] Athos. https://github.com/mpc-msri/EzPC/tree/master/Athos, June 2021.

[2] EzPC - a language for secure machine learning. https://github.com/mpc-msri/EzPC/tree/master/EzPC, June 2021.

[3] Secure and correct inference (SCI) library. https://github.com/mpc-msri/EzPC/tree/master/SCI, June 2021.



A Artifact Appendix

A.1 Abstract

Piranha is an open-source platform (https://github.com/ucbrise/piranha) for accelerating multi-party computation protocols using the GPU. Our artifact for Piranha consists of the prototype platform, along with scripts needed to replicate our experiments from Section 6. In particular, our artifact supports 3 LSSS protocols and a protocol-agnostic Neural Network training library. We demonstrate that such GPU acceleration can speed complex MPC applications like NN training by orders of magnitude compared to the CPU, achieves meaningful training results, and does so in a significantly more memory-efficient manner than prior work.

Our artifact is a combination of the Piranha platform and associated test scripts that can be used to validate these claims, specifically in replicating Figures 4-7 and Tables 2-4 in our paper, by executing the relevant micro- and end-to-end benchmarks. Piranha requires that each party participating in the protocol provision a machine with a GPU, along with access to the NVIDIA CUDA toolkit. The expected result of evaluation is to be able to replicate each major paper figure.

A.2 Artifact check-list (meta-information)

- **Program:** C++-based MPC GPU acceleration platform, with associated protocol-independent neural network library.
- **Run-time environment:** Cloud-based GPU cluster, with parties each executing on dedicated GPUs. A separate control server dispatches test runs to each party.
- **Execution:** Script-based for individual figure and table replication.
- **Metrics:** Computation and communication time, communication cost, training/test accuracy, and GPU memory footprint.
- **Output:** Data replicating paper figures and tables.
- **Experiments:** Secure training and inference passes, basic accelerated operation benchmarks, memory footprint measurement under various changes
- **How much time is needed to prepare workflow (approximately)?:** Provisioning a cluster of GPUs and a control server to run experiments might require 1-2 hours to set up.
- **How much time is needed to complete experiments (approximately)?:** Multi-party computation remains very slow compared to state-of-the-art plaintext. Expect almost no hands-on time but at least 40 hours of compute-time without evaluating the largest network (VGG16) and upwards of 400 compute-hours for VGG to replicate Figure 5.
- **Publicly available (explicitly provide evolving version reference)?:** Available at https://github.com/ucbrise/piranha/tree/main
- **Code licenses (if publicly available)?:** Piranha is licensed under the MIT License
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/ucbrise/piranha/commit/ddfb646f6f0e37e20194e4437e0d8e303fd89e4c

A.3 Description

The Piranha artifact consists of (1) the implementation of the device layer for integer-based GPU acceleration, (2) three LSSS protocols with varying party setups, (3) a neural network training library, and (4) a set of experimental runtime scripts to replicate the experiments we show in the manuscript.

For artifact evaluation, we assume a cloud-based cluster of compute- and GPU-provisioned machines, for MP-SPDZ and Piranha benchmarks, respectively. For LAN and WAN experiments, a replicator will provision 4 machines in a local (1ms latency) network and 2 additional machines in a different network (60ms latency), each acting as one party in a Piranha computation. Finally, a separate machine should act as a control server that runs the replication script.

A.3.1 Hardware dependencies

Piranha requires that each machine be provisioned with an NVIDIA GPU, and was initially evaluated on NVIDIA Tesla V100 GPUs.

A.3.2 Software dependencies

In a standalone installation, Piranha depends on installed GPU drivers, the NVIDIA CUDA Toolkit libraries, as well as CUTLASS and gtest.

A.3.3 Data sets

Piranha uses common ML datasets – MNIST, CIFAR10, and Tiny ImageNet; you may use scripts provided in the artifact to download and format the MNIST and CIFAR10 datasets we use for training.

A.3.4 Models

The artifact includes as part of the neural network training library a set of common neural network architectures in JSON format that can be executed using Piranha.

A.3.5 Security, privacy, and ethical concerns

N/A. All evaluation is done between parties that the evaluator directly controls; there is no interaction with third parties.

A.4 Installation

To install the artifact, clone the repository and follow the most up-to-date installation instructions at https://github.com/ucbrise/piranha. Install the required software dependencies, and, if using Piranha to perform training or inference, download the desired datasets with the provided scripts. Build the needed integer kernels with CUTLASS and compile the project for a given fixed point precision and multi-party protocol using the step-by-step instructions provided.

A.5 Evaluation and expected results

The primary evaluation script can be found at `experiments->run_experiment.py`, which will connect to and spawn execution on each of the GPU-provisioned parties as needed. Use the script to individually recreate figures and tables from the paper. You can choose to focus on the faster results first, leaving the extremely long VGG16-based results until you've verified the first set.

The main claims of the paper are that (1) GPU acceleration can improve MPC runtime in a protocol-agnostic manner far above CPU performance, (2) enabling realistic machine learning training, and (3) that Piranha accomplishes this while providing much better memory efficiency. Figure 4 and Table 4 support the first, showing a large improvement in runtime. Table 2 and Figures 5 and 6 show that real training results can be achieved in a reasonable amount of time, while Table 3 and Figure 7 demonstrate the benefits of Piranha's memory-conscious approach on memory footprint, thus achieving larger batch sizes. Reproducing these results involves running individual iterations or full training epochs for every table or figure in the paper, and comparing the output values or figures to those in the manuscript.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



F Artifact Appendix

F.1 Abstract

This demo was

- announced 2020.04.16 on the pqc-forum mailing list,
- updated 2020.04.23 from OpenSSL `1.1.1f` to OpenSSL `1.1.1g`,
- updated 2021.06.08 from OpenSSL `1.1.1g` to OpenSSL `1.1.1k`, including additional support for sntrup857,
- updated 2021.09.30 from OpenSSL `1.1.1k` to OpenSSL `1.1.1l`, alongside an update of the instructions to use `stunnel 5.60` and `glib-networking 2.60.4`, and
- updated 2021.11.02 to cover usage of `tls_timer` and suggestions regarding its use for experiments, and
- updated 2021.12.14 from OpenSSL `1.1.1l` to OpenSSL `1.1.1m`.

Our patches work for versions of OpenSSL from `1.1.1f` to `1.1.1m`.

This is a demo of OpenSSLNTRU web browsing taking just 156317 Haswell cycles to generate a new one-time `sntrup761` public key for each TLS 1.3 session. This demo uses: (i) the Gnome web browser (client) and `stunnel` (server) using (ii) a patched version of OpenSSL `1.1.1m` using (iii) a new OpenSSL `ENGINE` using (iv) a fast new `sntrup761` library.

The TLS 1.3 integration in OpenSSLNTRU uses the same basic data flow as the CECPQ2 experiment carried out by Google and Cloudflare. Compared to the cryptography in CECPQ2, the cryptography in OpenSSLNTRU has a higher security level and better performance. Furthermore, OpenSSLNTRU's new software layers decouple the fast-moving post-quantum software ecosystem from the TLS software ecosystem. OpenSSLNTRU also supports a second NTRU Prime parameter set, `sntrup857`, optimizing computation costs at an even higher security level.

F.2 Artifact check-list (meta-information)

- **How much time is needed to prepare workflow (approximately)?:** 60 min
- **How much time is needed to complete experiments (approximately)?:** 5–60 min
- **Publicly available?:** Y
- **Archived (provide DOI)?:** 10.5281/zenodo.5833729

F.3 Description

F.3.1 How to access

Visit `https://openssl ntru.cr.yp.to/demo.html`.

Additionally, we provide an archived version on Zenodo. The instructions in this appendix apply to the latter using, in place of the online URLs at `https://openssl ntru.cr.yp.to/`, the contents extracted from the downloaded archive.

F.3.2 Hardware dependencies

1. AVX2 support

F.3.3 Software dependencies

1. Linux
2. OpenSSL `1.1.1`

F.4 Installation

`https://openssl ntru.cr.yp.to/demo.html`

F.5 Evaluation and expected results

We claim the artifact at `https://openssl ntru.cr.yp.to/demo.html` reproduces two of the paper claims.

F.5.1 Reaching applications transparently

Following the provided instructions, the artifact allows to reproduce Section 4.3. By the end of the demo, you should achieve:

1. Setting up a TLS server with a custom TLS 1.3 cipher suite supporting `sntrup`.
2. Setting up a TLS client with a custom TLS 1.3 cipher suite supporting `sntrup`.
3. The user sees this upon issuing the last command `epiphany https://test761.cr.yp.to` as listed in the demo instructions.

The server side is optional, depending on if you want to talk to your own webserver or `https://test761.cr.yp.to`.

F.5.2 Macrobenchmarks: TLS handshakes

Additionally, the last part of the artifact covers the use of `tls_timer` to measure the wall-clock execution time of sequential TLS connections using different TLS groups, as described in Section 4.4.

Using `tls_timer` you can evaluate that

1. our specialized batch implementation (provided via `engNTRU` by `libsntrup761`) is faster than the reference code included in the optional patch to embed support for `sntrup761` operations in `libcrypto`;
2. with the caveats mentioned in Section 4.4, we achieve new records, in terms of computational costs, when compared with `X25519` and `NIST P-256`, the fastest pre-quantum implementations of TLS 1.3 key-exchange groups included in OpenSSL `1.1.1`, while providing higher pre-quantum security levels and much higher post-quantum security levels against all known attacks.

F.6 Notes

To reproduce the results summarized in Figure 5 in terms of absolute values, you would have to replicate the setup described in terms of hardware in footnote 6, and also take care of setting up both systems as detailed in the paragraph directly above the footnote to avoid biases due to CPU contention. It should also be noted that 100 experiments for each group, each performing 8192 connections, will require several hours, and that for the entire duration of the experiment you should ensure low network traffic and that no other processes (automated updates and other scheduled processes in particular) are executed on the machines running the experiments.

A simpler alternative, that would prove consistent with the results presented in Figure 5 in terms of sorting the groups according to the average connections per second, but not necessarily in absolute values, would be to install the server side and the client side of the demo on the same host, and then use `tls_timer` over the loopback interface, lowering the number of sequential connections (e.g., to 1024) to reduce the execution time of each experiment.

In any case, it is important to take care of the details listed in Section 4.4 regarding disabling frequency scaling, Turbo boost, concurrent services, and scheduled processes, isolating physical cores exclusively to each of the 3 processes (i.e., `tls_timer`, `stunnel`, and `apache2`) involved in each experiment run, and disabling/reducing logging to console or files, in order to minimize external causes of noise and achieve consistent results.



A Artifact Appendix

A.1 Abstract

Our artifact consists of the study protocol with respect to the survey provided to respondents. Additionally, we include expanded details of our statistical evaluation. We provide two supplemental files for the replication and interpretation of the work in the paper "Caring about Sharing: User Perceptions of Multiparty Data Sharing" to appear at USENIX Security 2022. Note that the survey includes opening and closing text as per the University of Waterloo's office of research ethics. This study received approval under ORE #: 41762.

There are no additional hardware or software requirements beyond a computer to access the files online. Readers can review the conclusions of our paper using the expanded statistics with respect to the tests performed and the test statistic values. The provided survey can be used to replicate our results or to apply them to different population sets or for related privacy perception studies.

A.2 Artifact check-list (meta-information)

- **Security, privacy, and ethical concerns:** This study received approval under ORE #: 41762. Participants were able to quit the study at any point and still receive remuneration. They were informed of the bounds of the study before participating and informed of the privacy focus after the study.
- **Publicly available:** https://github.com/bkacsmar/CaringAboutSharing)
- **Archived:** https://github.com/bkacsmar/CaringAboutSharing/releases/tag/V1.Usenix.22

A.3 Description

A.3.1 How to access

All files can be accessed at https://github.com/bkacsmar/CaringAboutSharing. They can be read in their online form, selectively downloaded, or by git-cloning the repository.

A.3.2 Hardware dependencies

Not applicable.

A.3.3 Software dependencies

Not applicable.

A.3.4 Data sets

Not applicable.

A.3.5 Models

Not applicable.

A.3.6 Security, privacy, and ethical concerns

A.4 Installation

Not applicable.

A.5 Evaluation and expected results

- The overall acceptability of multiparty data sharing is lower for collaborations that are not reciprocal. The inclusion of a health company in non-reciprocal collaborations is even less
- Across user controls, preferences for consent vary the most between collaboration types, however, opt-in consent is, generally speaking, the most acceptable.

To reproduce our results, if a study was done that sampled the same population, after doing the statistical tests, the same results should be found. We provide the results of our statistical tests across all variables we measured and our results can be verified by reviewing the values of the test statistics.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

The artifacts available at https://github.com/IAIK/Jenny contain the source code of the prototype described in the paper including any instructions and scripts to reproduce the figures from the paper. Please see the included `README.md` for further instructions.

A.2 Artifact check-list (meta-information)

- **Program:** We use the existing tools `nginx` and `lmbench` for our benchmarks. Both are included as submodules within our artifact repository and compiled from source.
- **Run-time environment:** The artifacts require Ubuntu 20.04 with Linux 5.4.0 and root permissions. Detailed setup instructions are provided in the `README.md`.
- **Hardware:** A CPU with Memory Protection Keys (MPK) (e.g., Intel Xeon Scalable) is required.
- **Output:** The artifacts include scripts to generate all benchmark-related figures from the paper.
- **How much disk space required (approximately):** 10GB
- **How much time is needed to prepare workflow (approximately):** 1–4 hours
- **How much time is needed to complete experiments (approximately):** approx. 1 day
- **Publicly available:** https://github.com/IAIK/Jenny

A.3 Description

A.3.1 How to access

The artifacts are available at https://github.com/IAIK/Jenny/tree/39bb0c696ce3c178e9593b7dbc034b2447ba2d00. Instructions on how to clone and use them, as well as any required hardware and software dependencies are detailed in the `README.txt` within this repository.

A.4 Installation

The artifacts are built and installed separately for the different benchmarks. See instructions below.

A.5 Evaluation and expected results

When the current working directory is `code/OurLib`, then the microbenchmarks can be run with `make bench-x86`. This creates a new subdirectory called `benchmarks/output_syscalls_{datetime}` for the results. There, `tc_getpid.pdf` and `tc_open.pdf` should be created, which were used in the paper as Figure 4 and 5.

For the application benchmarks, first our library has to be recompiled using `make app-bench-x86`.

Then, within the `benchmarks` directory, the three commands `./run_all.sh applications`, `./run_all.sh nginx`, and `./run_all.sh lmbench` will place the results in the `benchmarks/output_{datetime}`. The resulting pdf files `appbench_single.pdf`, `lmbench_numbers.pdf`, `appbench_nginx_single.pdf`, and `output_true_initialization_overhead.pdf` correspond to Figures 6, 7, 9, and 10 from the paper.

Note, that the resulting numbers will be slightly different compared to the paper since they are dependent on the exact CPU model, CPU frequency, kernel version, compiler versions, virtualization, etc.



A Artifact Appendix

A.1 Abstract

The artifact reproduces the results shown in Section 5 and the exploits showcased in Section 6. More specifically, we provide code to: (i) test if a system is vulnerable to BHI, (ii) verify if out-of-place BTI is possible, (iii) validate the results in Table 3, (iv) and verify the two exploits (inter- and intra- mode). The artifacts for x86-64 have been validated on Intel Core i7-10700K and Xeon Silver 4310 running Ubuntu 20.04 with Linux kernel 5.14, while the Arm results have been verified on the performance cores of a Google Pixel 6 (Cortex X1). All our source code is available on GitHub at `https://vusec.net/projects/bhi-spectre-bhb`
Following is the directory tree of the artifact:

```
bhi-spectre-bhb
├── re
│   ├── x64
│   │   ├── bhi_test          (Section 5.1)
│   │   ├── bhb_brute_force   (Section 5.2)
│   │   ├── bhb_size          (Section 5.3)
│   │   └── bhb_control       (Section 5.3)
│   └── arm
│       ├── bhi_test          (Section 5.1)
│       └── bhb_brute_force   (Section 5.2)
├── pocs
│   ├── inter_mode            (Section 6.2)
│   └── intra_mode            (Section 6.3)
└── tools
```

A.2 Artifact check-list (meta-information)

- **Experiments:** We provide self contained experiments matching the results of specific sections.
- **Compilation:** gcc, aarch64-linux-android31-clang, nasm.
- **Binary:** One binary per experiment in each directory.
- **Run-time environment:** For x86-64 experiments: Ubuntu 20.04 with Linux kernel 5.14. We provide the default Ubuntu kernel `.config` file (at the time of writing) on GitHub. For arm experiments: Android 12 with kernel 5.10. For both architectures, `bhi_test` uses a customized kernel.
- **Hardware:** x86-64 results were validated on Intel Core i7-10700K and Xeon Silver 4310. Arm results were validated on a Google Pixel 6.
- **Run-time state:** Set Linux CPUFreq governor to `performance`.
- **Execution:** Each folder contains a `./run.sh` script to run the experiment. When additional steps are required, this is specified in the README.
- **Output:** Each experiment provides only textual output. We describe in details the expected outcome for each experiment in Section A.6 and in the READMEs available in the corresponding directory.
- **How much disk space required (approximately)?:** 8GB are sufficient if the experiments are run using provided kernel images. Otherwise 80GB are needed.
- **How much time is needed to prepare workflow (approximately)?:** Few minutes in total. Each experiment and their corresponding environment can be set up with a single `./run.sh` bash script.
- **How much time is needed to complete experiments (approximately)?:** Approximately 5 minutes per experiment to run and verify the results of each experiment in `re/` and in `pocs/`.
- **Publicly available (explicitly provide evolving version reference)?:** All the source code is available at `https://vusec.net/projects/bhi-spectre-bhb`.
- **Code licenses (if publicly available)?:** Apache License 2.0.
- **Archived (explicitly provide DOI or stable reference)?:** The `ae_final` tag contains the final stable artifacts. Available at `https://github.com/vusec/bhi-spectre-bhb/releases/tag/ae_final`.

A.3 Description

A.3.1 How to access

All the source code is available at `https://github.com/vusec/bhi-spectre-bhb/releases/tag/ae_final`. Use the version under the tag *ae_final* for reproducing these results.

A.3.2 Hardware dependencies

The experiments in `re/x64/` were tested on all the Intel CPUs in Table 2. These also run on AMD, however they will not yield any interesting result since these systems are not vulnerable. The experiments in `re/arm/` were validated on a Google Pixel 6. The two end-to-end exploits (`pocs/`) were tested against the Intel Core i7-10700K and Xeon Silver 4310. Some adjustments to the cache eviction strategies and timings may be required on different Intel CPUs.

A.3.3 Software dependencies

We rely on standard build tools available in the Ubuntu package manager: `build-essentials`, `nasm`, `debootstrap` and `qemu-system-x86`. We also rely on `msr-tools` to read the `msr` specifying the availability of IBRS and eIBRS in a system. For the `bhi_test` experiments a modified Linux kernel is required. The kernel image and the source patch file are available as part of the artifact.

A.4 Installation

You can build all the artifacts from their corresponding directory using the following command depending on the target:

```
make UARCH=INTEL_10_GEN | INTEL_11_GEN | PIXEL_6
```

The only exception are `bhi_test` experiments. In order to run these, you need to first set up a VM with a custom Linux kernel (x86-64), or install a customized kernel directly (Arm). The kernel images are available as part of the artifact, as well as the patch file required to compile the kernel from source with our modifications. For x86-64, you can set up and start the VM in a few minutes following the instructions in the README found inside the `re/x64/bhi_test/vm` directory, while for Arm it is sufficient to boot the image using `fastboot boot boot.img`.

A.5 Experiment workflow

You can then execute every experiment by simply executing the `./run.sh` script in each directory.

A.6 Evaluation and expected results

In our work we make three main claims: (i) We show how Intel eIBRS and Arm CSV2 are incomplete solutions against cross-privilege BTI attacks, and introduce *Branch History Injection* (BHI) as a new primitive to build such attacks; (ii) We leverage BHI to build an end-to-end exploit on Intel systems deploying eIBRS (i.e., inter-mode); (iii) And we show that even when cross-privilege history injection is not possible kernel-to-kernel exploits (i.e., intra-mode) are still practical.

The experiments in the `re/` directory are meant to validate claim (i) for both x86-64 and Arm architectures.

The two end-to-end exploits in the `pocs/` directory are meant to validate claims (ii) and (iii).

We now describe the goal and expected output for each of these experiment. More details are available in the READMEs in each folder. The first experiments are meant to verify the claims on Intel CPUs.

- **(x64) bhi_test.**
  - *Goal.* Verify if the system is vulnerable to BHI.
  - *Implementation.* As described in Algorithm 1.
  - *Results.* On vulnerable systems we expect F+R to provide a hit rate $> 85\%$.
- **(x64) bhb_brute_force.**
  - *Goal.* Verify if we can carry out out-of-place BTI.
  - *Implementation.* As described in Figure 4, we use two different call sites and randomize the preceding jump chains.
  - *Results.* On vulnerable systems we expect stable collisions (F+R hit rate $> 85\%$) and 2^{14} iterations on average before finding a collision on Intel 10th gen CPUs—the iterations become 2^{17} for Intel 11th gen.
- **(x64) bhb_size.**
  - *Goal.* We want to recover the number of branches the BHB can keep track of.
  - *Implementation.* As described in Figure 5.
  - *Results.* We should observe predictions for $n = 29$ and $n = 66$ on the Intel Core i7-10700K and Xeon Silver 4310 respectively.
- **(x64) bhb_control.**
  - *Goal.* We want to recover the minimum number of branches under control by the attacker to generate arbitrary BTB collisions.
  - *Implementation.* As described in Figure 7.
  - *Results.* We should observe collisions for $k = 9$ and $k = 8$ on the Intel Core i7-10700K and Xeon Silver 4310 respectively.
- **(Arm) bhi_test.**
  - *Goal.* Verify if the system is vulnerable to BHI.
  - *Implementation.* As described in Algorithm 1.
  - *Results.* On the Cortex X1 we expect F+R to provide a hit rate $> 90\%$.
- **(Arm) bhb_brute_force.**
  - *Goal.* Verify if we can carry out out-of-place BTI.
  - *Implementation.* As described in Figure 4, we use same or different call sites and randomize the preceding jump chains.
  - *Results.* On the Cortex X1 we expect stable collision (F+R hit rate $> 90\%$) for in-place BTI, while no collision for out-of-place BTI.
- **(PoC) inter_mode.**
  - *Goal.* Showcase an end-to-end exploit leveraging BHI to perform cross-privilege mistraining and eBPF to read arbitrary kernel memory.
  - *Implementation.* As described in Section 6.2.
  - *Results.* It should take less than a minute to build an eviction set and then start leaking kernel memory.
- **(PoC) intra_mode.**
  - *Goal.* Showcase an intra-mode exploit where we take advantage of eBPF to perform both mistraining and misprediction.
  - *Implementation.* As described in Section 6.3.
  - *Results.* It should take less than a minute to build an eviction set and then start leaking kernel memory.

A.7 Experiment customization

Our reverse engineer programs, as well as our exploit, can also be run on different hardware with a suitable configuration. In particular, the tool `fr_checker` can be used to find the correct F+R threshold, and the file `common/targets.h` to specify the microarchitectural parameters.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



D Artifact Appendix

D.1 Abstract

The artifact reproduces the reverse engineering experiments outlined in §4 and §A with results summarized in Table 1 and Table 2, the computation of optimized eviction sets presented in §5, as well as the case studies discussed in §6. Specifically, we provide 3 code trees: `TLB/` contains Linux kernel modules used for reverse engineering TLB properties, along with helper userspace programs; `cache-ninja/` models set-associative caches with replacement policies and computes optimized eviction sets; `case-studies/` contains the case study experiments. The hardware used for `TLB` is detailed in Table 1 and Table 2, whereas the case studies were developed for Intel Kaby Lake processors. Finally, `cache-ninja` is architecture independent and can run on any computer. All our source code is available at https://github.com/vusec/tlbdr.

D.2 Artifact check-list (meta-information)

- **Compilation:** `cache-ninja` requires a rust compiler and cargo version $\geq$ 1.49. `TLB` and `case-studies` work with the system gcc.
- **Run-time environment:** the kernel modules in `TLB` and `TLB/AMD` are programmed against kernel versions 5.4 through 5.18. The kernel module in `TLB/PCID` assumes kernel version 5.4. Module insertion requires root access and a policy allowing loading of unsigned modules.
- **Hardware:** `case-studies` require an Intel Kaby Lake i7-7700 CPU with Hyperthreading enabled. Other Kaby Lake CPUs might be usable with small tweaks. `TLB` runs on the architectures shown in Table 1 and Table 2.
- **Execution:** The reverse engineering experiments under `TLB` are best run on a quiescent system, or at the very least one idle core. The experiments under `case-studies` are best run pinned on idle cores, ideally enforced via e.g., cpusets.
- **Output:** Each piece of code produces bespoke output, usually textual, representing its results. We describe this output in more detail in the README file of each directory, and provide tools to process this output.
- **Experiments:** The README file under each directory provides instructions on how to set up and run each experiment. A convenience script and/or Makefile is also included.
- **How much disk space required (approximately)?:** a few MiB for source and build.
- **How much time is needed to prepare workflow (approximately)?:** a few minutes for each experiment, mostly for setting up environment.
- **How much time is needed to complete experiments (approximately)?:** runtime usually depends on the number of measurements taken, which can be adjusted; by default the reverse engineering experiments in `TLB` and `TLB/PCID` run within 2–3 minutes, the experiments under `TLB/AMD` may take up to 30 minutes, `cache-ninja` runs within a minute, `case-studies/anc` takes up to one hour, while `case-studies/pthammer` and `case-studies/tlbleed` take around 10 minutes each.
- **Publicly available (explicitly provide evolving version reference)?:** All the source code is available at https://github.com/vusec/tlbdr.
- **Code licenses (if publicly available)?:** The kernel modules under `TLB` and `case-studies/pthammer/ptsim` are licensed GPLv2, the rest of the code is licensed under Apache 2.0.
- **Archived (explicitly provide DOI or stable reference)?:** The `sec22-ae-final` git tag marks the tree with the artifacts submitted for evaluation.

D.3 Description

D.3.1 How to access

All the source code is available at https://github.com/vusec/tlbdr, under the tag `sec22-ae-final`.

D.3.2 Hardware dependencies

- `TLB`: The experiments in `TLB` and `TLB/PCID` were run on the CPUs listed in Table 1, while the experiments under `TLB/AMD` were run on the CPUs listed in Table 2.
- `case-studies`: The experiments make microarchitectural assumptions that require an Intel Kaby Lake CPU and were run on an i7-7700K, as described in more detail in §6. A different model within the same family should also work, but might require some manual parameter tuning.

D.3.3 Software dependencies

- `TLB`: Building the kernel module requires kernel headers and system build tools. Running `prepare.sh` will install these dependencies on Ubuntu systems.
- `cache-ninja`: Building and running require rust and cargo version $\geq$ 1.49; earlier versions may work, although not tested.
- `case-studies`: Building and running requires a C compiler and, depending on the experiment, kernel headers and Python.

D.4 Installation

- `TLB`: Run `make` to build the kernel module and `insmod mmuctl/mmuctl.ko` to insert the kernel module (as root). To run the `PCID` or `AMD` experiments, navigate to the corresponding subdirectory before running the commands.
- `cache-ninja`: Run `cargo build [--debug|--release]` to compile the binary. Internet access might be required for cargo to download dependencies.
- `case-studies`: Run `make` in each case study subdirectory to build the experiment.

D.5 Experiment workflow

- `TLB`: Start with a freshly booted Linux system running on bare metal and ensure the build dependencies are satisfied. Run `make test`; this will build the experiment kernel module, load the module and execute the trigger binary. If the kernel module is already inserted it sufficies to run the trigger binary directly. After the experiment run `make unload` to remove the module from the kernel. To run the `PCID` or `AMD` experiments, navigate to the corresponding subdirectory before running the commands. Warning: these experiments run kernel code that might crash/hang a core or otherwise leave the kernel in an invalid state, we recommend you monitor the kernel log and immediately reboot your system after any error.
- `cache-ninja`: Run `cargo run` to build and run the program, which then computes and prints the optimized eviction sets discussed in §5.
- `case-studies`: Execute `make run` in each directory to run experiments with default settings. Consult the individual README files under each directory for detailed instructions.

D.6 Evaluation and expected results

In this work we make several main claims: (i) We introduce TLB desynchronization as a reliable primitive for TLB reverse engineering and validate it against previous work; (ii) We show how TLB desynchronization, due to its precision and reliability, can be used to reverse engineer previously undocumented TLB features; (iii) We show how knowledge of one of these properties—replacement policies—enable knowledgeable manipulation of TLB state, leading to vastly more efficient adversarial evictions; and (iv) We show how these more efficient adversarial evictions bring significant improvements to various classes of attacks that make use of TLB eviction.

- `TLB` is the reverse engineering code using TLB desynchronization which supports claims (i) and (ii).
- `cache-ninja` implements a model of TLB state along with the replacement policies that we reverse engineered, and uses this model to compute optimal adversarial eviction sets, validating claim (iii).
- `case-studies` implements proofs-of-concept for integrating the previously computed optimal eviction sets into several existing attacks, in support of claim (iv).

We now describe the goal and expected output of each of our experiments.

- `TLB`
  - *Goal:* Measure TLB properties.
  - *Implementation:* As described in §4 and §A.
  - *Results:* On the relevant systems, we expect results in line with Table 1 and Table 2.
- `cache-ninja`
  - *Goal:* Produce optimal eviction sets for a Kaby Lake TLB
  - *Implementation:* As described in §5, using BFS to search through the state graph.
  - *Results:* We expect the optimized eviction sets as presented in §5.2, §5.3, and §6.2.
- `case-studies/anc`
  - *Goal:* Measure and compare the speed of AnC attacks using naive and optimized eviction sets.
  - *Implementation:* As described in §6.1.
  - *Results:* On relevant hardware we expect results comparable to Figure 7.
- `case-studies/pthammer`
  - *Goal:* Reproduce Figure 8 from raw data; optionally produce a histogram of all data.
  - *Implementation:* As described in §6.1.
  - *Results:* We expect a faithful rendering of Figure 8. The histogram should also show distinctly tri-modal output, as described in §6.1.
- `case-studies/pthammer/ptsim`
  - *Goal:* Simulate, measure and compare the potential hammer rate of a PTHammer-like attack using naive and optimized eviction sets.
  - *Implementation:* As described in §6.1.
  - *Results:* On relevant hardware we expect results similar to those described in §6.1 and shown in Figure 8.
- `case-studies/tlbleed`
  - *Goal:* Measure and compare the speed of TLBleed-style attacks using naive or optimized eviction sets.
  - *Implementation:* As described in §6.2.
  - *Results:* On relevant hardware, we expect raw sample rates in line with those shown in Table 5. Similarly, we expect peak sustained covert channel bandwidth in line with that described in §6.2.

D.7 Experiment customization

Customization and tweaking of experiments is possible to some degree, consult the individual README files of each subdirectory for more details.

D.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

This artifact provides binaries and OS scripts to evaluate a Bluetooth over-the-air fuzzer under a PC (x86_64) running Ubuntu 18.04. Due to its over-the-air approach and dependency on Bluetooth target devices, access to a remote machine is provided via SSH with private key. Moreover, we design six experiments to assist in replicating the main results of the paper by generating figures and terminal outputs after the fuzzing campaign ends. The evaluation procedure consists of OS scripts that are either included in the artifact or described in this appendix. The results generated by our experiments will help support the claims that (i) our fuzzer outperforms other state-of-the-art over-the-air BT fuzzer, (ii) that our internal fuzzing components are essential and add to the effectiveness of the fuzzer and (iii) that our fuzzing framework is extensible to other wireless protocols beyond Bluetooth such as Wi-Fi and BLE. Lastly, the artifact also includes exploits to launch against real wireless devices (BT, Wi-Fi and BLE) attached to a remote machine.

A.2 Artifact check-list (meta-information)

Obligatory. Fill in whatever is applicable with some keywords and remove unrelated items.

- **Algorithm:** Braktooth OTA Fuzzing
- **Compilation:** GCC version 7.5.0 for modules compilation, fuzzer binaries provided in artifact, source code upon request.
- **Binary:** bt_fuzzer, wdmapper (included with artifact)
- **Run-time environment:** Ubuntu 18.04, Kernel 5.11.13
- **Hardware:** ESP-WROVER-KIT, ESP32-Ethernet-KIT, Oppo Reno 5G, Raspberry 3B and x86_64 Computer
- **Metrics:** Execution Time, Model Coverage, Number of Crashes, Number of Anomalies
- **Output:** Console, files (.txt, .csv) and graphs.
- **Experiments:** Os scripts and manual steps by the user.
- **How much disk space required (approximately)?:** 4 GB
- **How much time is needed to prepare workflow (approximately)?:** 10 min.
- **How much time is needed to complete experiments (approximately)?:** 30 hours.
- **Publicly available (explicitly provide evolving version reference)?:** https://doi.org/10.5281/zenodo.7023642

A.3 Description

The artifact showcases the capabilities of our systematic directed fuzzing framework that automatically discover implementation bugs in arbitrary Bluetooth Classic (BT) devices. We also showcase the flexibility of our approach, which can be applied to other wireless protocols such as Wi-Fi and BLE.

A.3.1 How to access

The access to the target Evaluation Machine can be done via SSH after the reviewer sends his **SSH public key** to the researchers during the artifact evaluation period.

Once access has been granted, the target Evaluation Machine can be accessed via SSH using linux/macos as follows:

```
ssh artifact@evaluation.braktooth.com -p 2222
```

If the reviewer cannot share his/her public SSH public key, we can send our SSH private key (artifact.key), which can be used to access the Evaluation Machine as follows:

```
chmod 0600 artifact.key
ssh -i artifact.key artifact@evaluation.braktooth.com -p 2\
    222
```

X11 forwarding is recommended to be enabled in the SSH client to visualize pdf figures. Otherwise, figure files can be transferred via SFTP.

To access the remote Evaluation Machine from windows, the software MobaXterm can be used as it has X11 enabled by default.

A.3.2 Hardware dependencies

The following hardware development boards are required to evaluate the fuzzer:

- ESP32-WROVER-KIT - Bluetooth Fuzzing Interface
- ESP32-Ethernet-KIT - Vulnerable Bluetooth/Wi-Fi Target
- Oppo Reno 5G - Vulnerable Bluetooth Target
- Raspberry 3B - Vulnerable Wi-Fi Target

All the listed hardware dependencies are connected to the remote Evaluation Machine.

A.3.3 Software dependencies

The software dependencies for the fuzzer runtime is provided in the artifact script *requirements.sh*. Such script is intended to be executed under Ubuntu 18.04. However, the main runtime dependencies are listed below:

- Wireshark 3.7.0 (Included with artifact)
- Python3 $\geq$ 3.6.9 (Included with artifact)
- Node.js v12.22.12

Furthermore, the vulnerable SDK (esp-idf commit 3de8b79) for the vulnerable target (ESP32) must be installed in the host pc to flash the vulnerable firmware to the target.

Lastly, for comparing our fuzzer with other BT OTA fuzzers, the following 3rd party software is required:

- Bluetooth Stack Smasher v0.6
- BlueFuzz
- bfuzz (iotcube) v2.2.0

Note that the above BT fuzzers are installed via their respective *modules/eval/experimen3-*.sh* scripts.

A.3.4 Data Sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

To avoid causing unintended malfunctions to arbitrary Bluetooth devices, the artifact must be only used against devices which are strictly authorized by the device's owner. Therefore, it is advisable to only fuzz the wireless targets discussed in the experiment workflow.

Furthermore, our remote Evaluation Machine is configured to not log any SSH connection to ensure privacy of the reviewer during the artifact evaluation period.

A.4 Installation

The installation of the fuzzer can be done by running few scripts provided in the artifact binary repository. Download and installation of the fuzzer and software dependencies can be done as follows:

```
mkdir braktooth
cd braktooth
wget https://zenodo.org/record/7023642/files/release.zip
unzip release.zip
# Install software requirements for Ubuntu 18.04
sudo apt install zstd
tar -I zstd -xf wdissector.tar.zst # Extract binary folder
cd wdissector
sudo ./requirements.sh
sudo ./requirements.sh doc # Ignore errors
```

Next, a vulnerable SDK version of our Bluetooth target (ESP32 esp-idf commit 3de8b79) needs to be installed. The following commands can download and install the vulnerable SDK on the remote machine:

```
git clone https://github.com/espressif/esp-idf
cd esp-idf
git checkout 3de8b79 # Vulnerable version of esp-idf SDK
./install.sh
cd ../
```

A.5 Experiment workflow

Figure 1 illustrates the relevant hardware setup in which the experiments are performed on the remote Evaluation Machine. In the following, we describe the experiment workflow which leverages our hardware setup. Note that the fuzzer relies on the BT Fuzzing Interface, which uses the same board model as our *Target 1*, but they are separate boards as illustrated in Figure 1.

First, we aim to evaluate the Bluetooth fuzzer by running it against *Target 1: ESP32-WROVER*. After the maximum number of fuzzing iterations is reached, the fuzzer will stop and generate log files which are used to analyze the results during the experiments via the python scripts provided in the artifact folder *modules/eval*.

As illustrated in Figure 2 (a), the workflow is designed to first evaluate the different variants of the fuzzer by changing its configuration parameters. After each variant is evaluated, the log folder

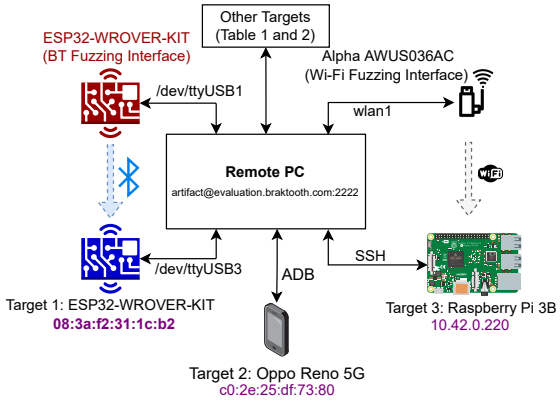


Figure 1: Main Hardware Setup of Evaluation Machine

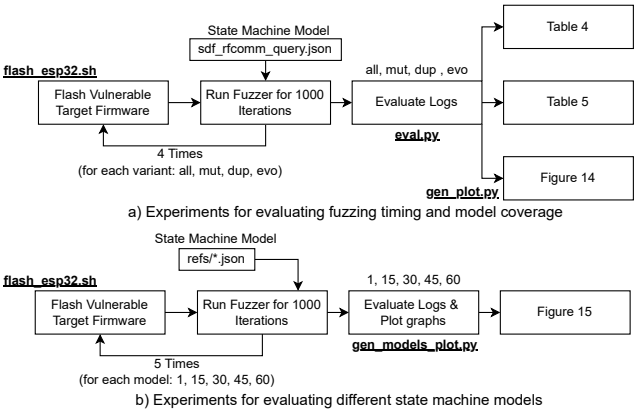


Figure 2: Diagram of Experiments Workflow

Table 1: All BT Devices Setup of Evaluation Machine

BT SoC Vendor	BT SoC	Dev. Kit / Product	BDAddress	Monitor
Bluetooth 5.2				
Intel	AX200	PCIe Module	6C:6A:77:53:97:2D	SSH artifact@127.0.0.1
Qualcomm	WCN399X	Oppo Reno 5G	C0:2E:25:DF:73:80	ADB 627ff0eb
Bluetooth 5.1				
Texas Instruments	CC2564C	CC256XCQFN-EM	98:5D:AD:12:03:F3	Serial /dev/ttyACM0
Bluetooth 5.0				
Cypress	CYW20735B1	CYW920735Q60EVB-01	20:73:5B:1C:D9:93	Serial /dev/ttyUSB6
Bluetrum Technology	AB5301A	AB32VG1	-	-
Zhuhai Jieli Technology	AC6925C	XY-WRBT Module	48:1B:B2:26:90:36	N.A
Actions Technology	ATS281X	Xiaomi MDZ-36-DB	-	-
Bluetooth 4.2				
Zhuhai Jieli Technology	AC6905X	BT Audio Receiver	1F:F6:F7:96:12:E3	N.A
Espressif Systems	ESP32	ESP-WROVER-KIT	08:3A:F2:31:1C:B2	Serial /dev/ttyUSB3
Bluetooth 4.1				
Harman International	JX25X	JBL TUNE500BT	-	-
Bluetooth 4.0				
Qualcomm	CSR 8811	Laird DVK-BT900-SA	-	-
Beken	BK3260N	HC-05	98:DA:60:00:42:E3	Serial /dev/ttyUSB10
Bluetooth 3.0 + HS				
Silabs	WT32i	DKWT32I-A	00:07:80:CC:7D:E3	Serial /dev/ttyUSB4

is fed to the script *eval.py*, which analyzes the packet trace file (*capture_bluetooth.pcapng*) and events logs (events.*.txt) to return timing, coverage and evaluation summary as depicted in Table 4 (Timing of 1000 fuzzing iterations for each device) and Table 5 (Evaluation summary w.r.t. different R_{sel} and D_T) of the paper.

Furthermore, the target is re-flashed via Espressif esp-idf SDK after each re-evaluation. This is to ensure a fresh state to the target in case its flash memory is corrupted during the fuzzing process. Next, after all the fuzzer variants are evaluated (*mutation*, *duplication evolution* and *all*), a graph similar to Figure 14 (crashes/deadlocks w.r.t ESP32 fuzzing iterations) is generated by running the script *gen_plot.py*.

Next, in Figure 2 (b), we illustrate the experiments to evaluate the state machine model of the fuzzer. Similar to the previous experiment, the fuzzer is evaluated multiple times, but with different state machine models. Then, the log results of each evaluation are analyzed by the python script *gen_models_plot.py* and a graph similar to Figure 15 in the paper is generated (Evaluation of different state machine models).

Further, Table 1, lists the BT devices attached to the Evaluation Machine, which can be used to evaluate the bugs and timing behavior depicted in Table 2 and Table 4 of the paper. The column *BDAddress* lists the target Bluetooth Address for fuzzing or exploitation, whereas column *Monitor* describes the monitor connection method with the target such as *Serial*, *SSH* or *ADB*. Moreover, Table 1 corresponds to Table 1 of the paper. However, four devices are currently not possible to remotely evaluate due to the following reasons:

- **Xiaomi MDZ-36-DB and JBL TUNE500BT** - Both BT products (Speaker and headphone respectively) turn off automatically when not receiving any BT connections and requires manual interaction to turn them on before a fuzzing session. Therefore, such devices are not possible to automate for the artifact.
- **Bluetrum AB5301A** - Such board has been updated with the latest proprietary firmware from vendor during the disclosure period, however, we have no copies of the older vulnerable firmware to ensure a proper evaluation with such device. We have contacted the vendor to acquire the older firmware, but we have not received a response so far.
- **Laird DVK-BT900-SA** - The board is non-functional due to a short circuit in the evaluation board which prevent further evaluation. Unfortunately, the DVK-BT900-SA development kit is out of stock as of the time of writing.

Note that the *Mic.* Monitor from the paper is indicated as *N.A* (not applicable) in Table 1 for devices XY-WRBT and BT Audio Receiver since the Evaluation Machine laboratory is in a noisy environment and outside our control. Therefore, the *"Microphone"* monitor is not evaluated in the artifact.

Finally, Table 2 lists the Wi-Fi and BLE devices connected to the Evaluation Machine. Column *"Address"* lists the BLE Address of each device, whereas it is *N.A* for Wi-Fi devices since they connect to the Wi-Fi AP fuzzer automatically. Such Table can be used to replicating Table 7 of the paper (*Summary of unknown flaws found by extension*).

Table 2: Wi-Fi / BLE Devices Setup of Evaluation Machine

Extension	Target	Address	Monitor
BLE Host	ESP32	08:3A:F2:31:1C:B2	Serial /dev/ttyUSB3
	Telink TLSR8258	A4:C1:38:D8:AD:A9	N.A
	NXP KW41Z	00:60:37:88:16:0C	Serial /dev/ttyACM2
	TI CC2540	38:81:D7:3D:45:A2	N.A
Wi-Fi AP	ESP32	N.A	Serial /dev/ttyUSB7
	ESP8266	N.A	Serial /dev/ttyUSB9
	Rasp. Pi 3 B	N.A	SSH pi@10.42.0.220
	One Plus 5T	N.A	ADB 3ffd4d9a

A.6 Evaluation and expected results

The results generated by our experiments will help support the claims that (i) our fuzzer outperforms other state-of-the-art over-the-air BT fuzzer, (ii) that our internal fuzzing components are essential and add to the effectiveness of the fuzzer and (iii) that our fuzzing framework is extensible to other wireless protocols beyond Bluetooth such as Wi-Fi and BLE.

Evaluation Instructions:

We start by flashing a vulnerable firmware into the target esp32 which is connected to the remote machine. A code snipped of the procedure is shown in Listing 1. For your convenience, such script is included in *modules/eval/flash_esp32.sh*.

Listing 1: Flashing firmware to ESP32 target (flash_esp32.sh)

```
cd esp-idf
source export.sh
cd examples/bluetooth/bluedroid/classic_bt/bt_spp_acceptor
idf.py build
# Program firmware to target (connected via /dev/ttyUSB3)
idf.py -p /dev/ttyUSB3 erase_flash flash
```

A.6.1 Experiment 1 - Evaluating Timing, Coverage and Fuzzing Components

This experiment is intended to run the fuzzer in different configurations to evaluate the components that contribute to the overall design of the fuzzer. The script included on *modules/eval/experiment1.sh* runs the fuzzer 4 times, switching between the fuzzing parameters *–mutation, –duplication, –optimization*.

The script below can be used to run experiment 1 for a ESP32 target with BDAddress of 10:52:1c:69:ac:82.

```
cd $HOME/braktooth/wdissector/modules/eval
./experiment1.sh
```

When running the script above, the terminal output illustrated in Figure 3 appears during the fuzzing session, indicating an exchange of over-the-air LMP packets between the fuzzing interface and the target *ESP32* device. Furthermore, upon end of evaluation (which takes several hours), extra logging folders and files are created as illustrated in Figure 4.

Now, we can start to analyze the outputs generated and relate to the tables and figures present in the paper. To start, we can get

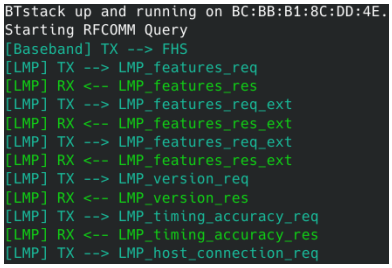


Figure 3: Expected output when the BT fuzzer is running

```
wdissector
└── modules
    └── eval
        ├── all ..... Logs for Variant 'All'
        ├── mut ..... Logs for Variant 'Mutation'
        ├── dup ..... Logs for Variant 'Duplication'
        ├── evo ..... Logs for Variant 'Evolution'
        └── graph_models.pdf .. Figure 14 in the paper
```

Figure 4: Generated folder and files after running *experiment1.sh*

the coverage and timing for the variant 'all', which corresponds to Table 4 in the paper, by running the following command in the *eval* folder:

Listing 2: Generate results for Table 4 (Timing of 1000 fuzzing iterations for each device)

```
cd $HOME/braktooth/wdissector/modules/eval
./eval.py all
```

The output of Listing 2 should look similar to Figure 5, thus returning relevant information that is present in Table 4 of the paper such as *Total Time, 1st Vulnerability, 1st Non-compliance* and *Model Coverage* (highlighted in blue). Although the time to complete 1000 iterations (*Total Time*) can variate, it is usually in the range of 3h-3:30h for a BT target such as ESP32. Nevertheless, *Model Coverage* for ESP32 has its value in the range $22-30\%$ for this evaluation, as exemplified in Figure 5. Moreover, due to the stochastic behavior of the over-the-air fuzzing process, the *1st Vulnerability, 1st Non-compliance* can variate significantly. Depending on the iteration, the first Vulnerability or Non-Compliance can be achieved in less than 1 minute in an optimistic scenario or after dozens of minutes, or almost one hour in worst case.

This experiment to generate *Table 4* mainly focuses on ESP32. However, customization of the experiment for evaluation of other BT targets is discussed in section A.7.1.

Next, the first entry of Table 5 of the paper (Evaluation summary w.r.t. different R_{sel} and D_T) is obtained by running the script of Listing 3 and getting the output of *"Evaluation Summary"* as illustrated in Figure 6. Note that the python script *eval.py* receives "*dup*" as argument, which refers to the *Duplication* fuzzing variant log folder, which was generated after running *experiment1.sh*.

Listing 3: Generate tesults for Table 5 (Evaluation summary w.r.t. different R_{sel} and D_T)

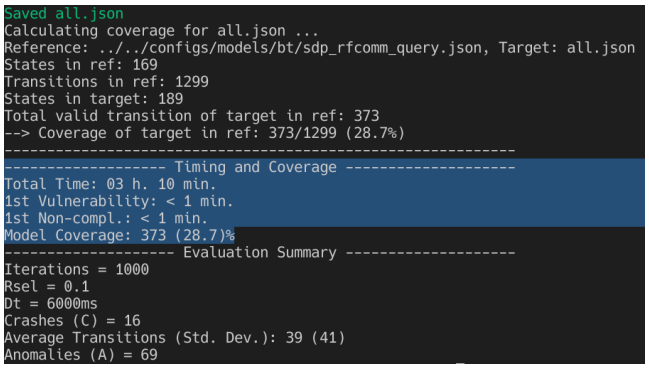


Figure 5: Example output for Table 4 results (ESP32 target)

```
cd $HOME/braktooth/wdissector/modules/eval
./eval.py dup
```

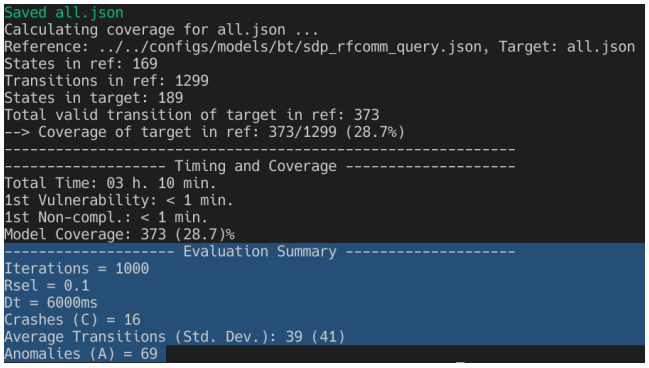


Figure 6: Example output for an entry of Table 5

It is worthwhile to mention that the generated result for this experiment is based on 1000 iterations instead of 200 iterations as described on the paper to avoid running an additional evaluation. Furthermore, since it requires a total of 9 evaluation to generate all entries of Table 5 as shown in the paper, this experiment only focuses on the first one to save time during this experiment. Nevertheless, generating more entries or limiting the number of iteration can be done by changing certain configuration parameters as later discussed in the Experiment Customization (Section A.7.1).

Furthermore, the output obtained for Listing 3 shall indicate more *Crashes (C)* than indicated on the paper due to the extended maximum number of iterations, which results in more time to find crashes. On the other hand, the *Average Transitions (Std. Dev.)* should stay relatively the same as indicated on Table 5 in the paper (107 ± 81 for entry $R_{sel} = 0.1$ and $D_T = 6000$).

Lastly, we can generate a figure similar to the Figure 14 presented in the paper (crashes/deadlocks w.r.t ESP32 fuzzing iterations), albeit not for *unique crashes/deadlocks*, but rather for all reported crashes from the fuzzer. This is because our framework cannot automatically detect the root cause of each reported crash. Instead, the uniqueness shown on Figure 14 in the paper, requires manual and careful analysis of the target trace output. Automation of such effort to investigate the root cause is beyond the scope of our fuzzing tool.

Nevertheless, a figure for crashes/deadlocks w.r.t ESP32 fuzzing iterations is generated and opened by running the script below:

```
cd $HOME/braktooth/wdissector/modules/eval
./gen_plot.py
# Graph saved to graph_optimization.pdf.pdf
```

In the case the figure failes to open to your view, you can manually copy the figure locally via SFTP or call *okular* on the remote host machine. The latter approach requires X11 enabled in your SSH client:

```
okular graph_optimization.pdf
```

Figure 7 depicts a sample of the expected graph for this evaluation.

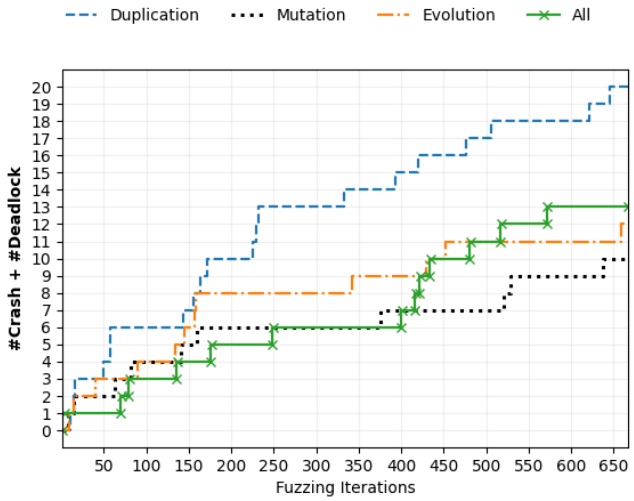


Figure 7: Sample graph for crashes/deadlocks w.r.t ESP32 fuzzing iterations.

A.6.2 Experiment 2 - Evaluating State Machine Model

As illustrated in the diagram of Figure 2 (b), this second experiment focuses in evaluating the differences in coverage, number of crashes and anomalies for different reference models used during the Bluetooth fuzzing session. To this end, the script *modules/eval/experiment2.sh* has been prepared to automate the generation and selection of the models before the evaluation starts. The relevant files used for the model generation are depicted in Figure 8.

This experiment already provides such reference capture to simplify the evaluation, however, Section A.7.2 details how to create clean reference captures that can be used to create reference models.

After running the script of Listing 4, you should get (after several hours) the terminal output depicted in Figure 9 and the evaluation graph of all the reference models as illustrated in Figure 10.

Listing 4: Generate results for Figure 15 (Evaluation of different state machine models)

```
cd $HOME/braktooth/wdissector/modules/eval
./experiment2.sh
# Graph saved to graph_models.pdf
okular graph_models.pdf
```

Figure 9 and Figure 10 relates to Figure 15 of the paper and depicts the number of states, model coverage, number of crashes

```
wdissector
└── modules
    └── eval
        └── refs
            ├── 1.pcapng  ..... 1 min. Reference capture
            ├── 15.pcapng  .... 15 min. Reference capture
            ├── 30.pcapng  .... 30 min. Reference capture
            ├── 45.pcapng  .... 45 min. Reference capture
            └── 60.pcapng  .... 60 min. Reference capture
```

Figure 8: Reference capture files used for reference model generation.

```
-------------------- Graph Summary ----------------------
====== eval_1 ======
Max iterations: 1000
Crashes: 10
Anomalies: 62
Coverage: 80.1
States count: 82
LMP states: 62
Non-LMP states: 20
====== eval_15 ======
Max iterations: 1000
Crashes: 7
Anomalies: 63
Coverage: 61.8
States count: 88
LMP states: 68
Non-LMP states: 20
====== eval_30 ======
Max iterations: 1000
Crashes: 8
Anomalies: 63
Coverage: 59.0
States count: 93
LMP states: 73
Non-LMP states: 20
====== eval_45 ======
Max iterations: 1000
Crashes: 5
Anomalies: 63
Coverage: 55.1
States count: 101
LMP states: 81
Non-LMP states: 20
====== eval_60 ======
Max iterations: 1000
Crashes: 9
Anomalies: 59
Coverage: 48.2
States count: 107
LMP states: 87
Non-LMP states: 20
------------------------------
Figure saved as graph_models.pdf
```

Figure 9: Sample terminal output of evaluation of different state machine models.

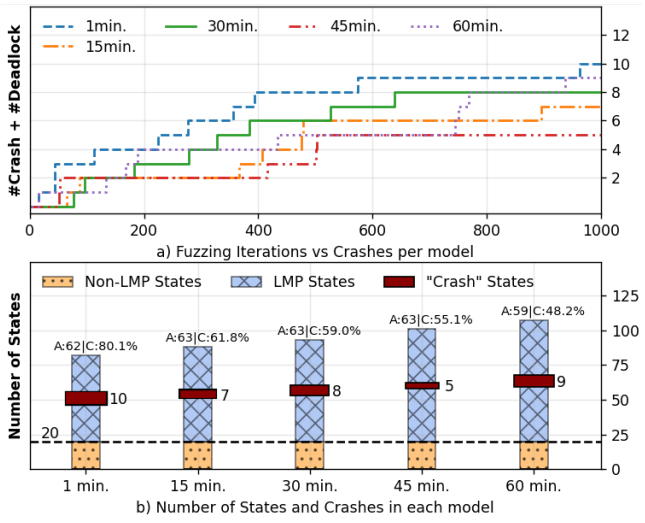


Figure 10: Sample graph of evaluation of different state machine models (graph_models.pdf).

and anomalies for each of the five reference models evaluated as described in the paper ($\{M^i_{ref} \mid i \in \{1, 15, 30, 45, 60\}\}$).

Overall, the number of states should slightly increase according to the model with the highest training time (M^{60}_{ref}) while the coverage should decrease for such model. This is because with a more complete reference model such as M^{60}_{ref}, more states are to be explored during the 1000 fuzzing iterations, which translates to a lower coverage as compared to a simpler reference model such as M^{1}_{ref}.

Customization of this experiment on how to generate reference captures from scratch is discussed in Section A.7.2.

A.6.3 Experiment 3 - State Mapping Generation

The artifact includes several reference capture files from protocols beyond BT Classic to evaluate the state mapper. However, in order to evaluate the state mapper, we can run the script of Listing 5 to generate the complete state machine visualization of the simplified graph presented in Figure 16 of the paper (An illustration of a simplified BT state machine and corresponding state mapping rules for LMP and L2CAP.). The state machine generation takes as input a reference capture (*capture_bt_a2dp.pcapng*) and the configuration file with the mapping rules (*config_bt.json*).

Figure 11 illustrates the generated state machine graph for the sample BT capture (*capture_bt_a2dp.pcapng*) and should correspond to Figure 16 of the paper.

Listing 5: Run state mapper for sample capture files (Figure 16 of the paper).

```
cd $HOME/braktooth/wdissector/examples/wdmapper/
# This will generate states_bt_a2dp.svg
./run_example_wdmapper.sh
sudo npm install svg2pdf -g
svg2pdf states_bt_a2dp.svg # Convert svg to pdf
okular states_bt_a2dp.pdf # Open pdf
```

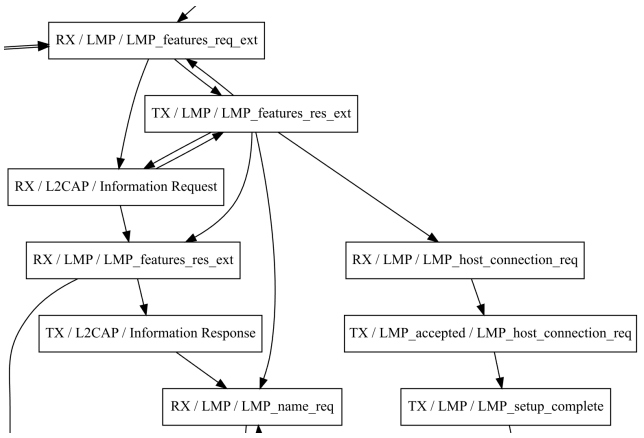


Figure 11: Simplified BT state machine and corresponding state mapping rules for LMP and L2CAP (states_bt_a2dp.svg).

A.6.4 Experiment 4 - Comparison between different fuzzing tools

On this experiment, we evaluate other BT OTA fuzzer against our ESP32 target and compare the results to Table 6 (A Comparison among different fuzzing tools).

For this experiment, we don't validate the entry *toothpicker* since it requires a special hardware setup which is outside the scope of our remote evaluation platform.

Nevertheless, the other third parties fuzzing tools (*bfuzz, Stack Smasher and Bluefuzz*) are installed and executed by running the following scripts:

```
cd $HOME/braktooth/wdissector/modules/eval
./experiment3_bss.sh # Stack Smasher
./experiment3_bfuzz.sh # bfuzz (iotcube)
./experiment3_bluefuzz.sh # Bluefuzz
```

While the first script (*bss*) does not require user interaction, the other scripts (*bfuzz and bluefuzz*) require the user to select the Bluetooth device before starting the fuzzing session. To this end, once such script are executed and a prompt asking for device number is requested, the user needs to select the number for device name "ESP_SPP_ACCEPTOR" and its respective BT service options as illustrated in Figure 12 and Figure 13 for the fuzzers *bfuzz* and *Bluefuzz* respectively.

It is worthwhile to mention that the scripts for this experimented are not completely automated, so the evaluator would need to press the keys *CTRL + C* to interrupt the fuzzing session after 3 hours for each script.

For all scripts, once a crash has been triggered, the terminal output should show the crash indication message for each fuzzer such as "Crash detected". Nevertheless, if the target (ESP32) becomes unresponsive during the session, the experiment script can be re-run to reset the target firmware.

It is expected that only *bfuzz (iotcube)* is able to trigger a crash in ESP32, however due to the random nature of the other fuzzers (*bss, Bluefuzz*), receiving a crash during the 3 hours evaluation period is still possible. However, we claim that our fuzzer outperforms the *state-of-the-art* (at the time of writing the paper) by finding new

bugs and non-compliance in the LMP layer of ESP32.

```
[+] Start scanning bluetooth devices...

        Target Bluetooth Device List
        [No.]   [BT address]            [Device name]
        00      08:3A:F2:31:1C:B2       ESP_SPP_ACCEPTOR
        01      5C:66:6C:6C:AC:98       OPPO Reno2 Z
        02      FC:77:74:9E:00:97       DESKTOP-Q3P21MS
        03      94:65:2D:ED:34:1F       OnePlus 5T
        Total : 4

select device: 00

Start scanning services...

        List of profiles for the device
        00      [0x1101]: Serial Port
        01      [0x0000]: SDP
        Total : 2

Select a profile to fuzz: 00
```

Figure 12: bfuzz BT device options screen.

```
Chose a device number for pairing (q for quit):0
You have chosen device 0: 08:3A:F2:31:1C:B2(ESP_SPP_ACCEPTOR)
Chose the service number :0
protocol: RFCOMM
port:  1
```

Figure 13: Bluefuzz BT device options screen.

Customization of this experiment on running the comparison against other BT devices of Table 1 is discussed in Section A.7.3.

A.6.5 Experiment 5 - Attacks Exploiting BrakTooth

In this experiment, we reproduce certain BT attacks against ESP32 as reported in the paper. We also provide an example of launching an attack against *Oppo Reno 5G* via the SSH monitor included in the fuzzer. Note that in this experiment we use ***Oppo Reno 5G*** instead of *Pocophone F1* as used in the paper due to unavailability of our *Pocophone F1* during the evaluation artifact period. Nevertheless *Oppo Reno 5G* uses the same BT SoC (WCN399X) as *Pocophone F1* and therefore is vulnerable to the same BT attacks.

Before launching the attack, we need to know the BDAddress of the target BT device. To facilitate this, BT Exploiter can scan the BDAddress of targets nearby by running the following command:

```
sudo bin/bt_fuzzer --scan
```

If ESP32 is detected, then you should get a similar output as shown in Figure 14.

```
Serial port /dev/ttyUSB3@4000000 opened
BT Scanning Started (Inquiry)...
[ESP32BT] BDAddress: 94:65:2d:ed:34:1f, Name: OnePlus 5T, RSSI: -54, Class: Smartphone
[ESP32BT] BDAddress: 08:3a:f2:31:1c:b2, Name: ESP_SPP_ACCEPTOR, RSSI: -30, Class: Unkown
[ESP32BT] BDAddress: fc:77:74:9e:00:97, Name: DESKTOP-Q3P21MS, RSSI: -42, Class: Desktop workstation
```

Figure 14: BT Scan output

Next, we can choose an exploit by its name and use the target BDAddress. For this example, evaluation, we start by launching the remote code execution attack against ESP32 as described in the paper (CVE-2021-28138) by running the following command:

```
sudo bin/bt_fuzzer --no-gui --exploit=\
     invalid_feature_page_execution
--target=08:3a:f2:31:1c:b2 --target-port=/dev/ttyUSB3
```

If the attack is successful, the fuzzer output should log the crash trace of the target ESP32 with a program counter (PC) set to 0*xdeadbeee*. Thus, indicating that we have control over the target's program counter.

```
[LMP] TX --> LMP_features_res_ext
[L2CAP] RX <-- Connection Response - Success (SCID: 0x0041, DCID: 0x0040)
[L2CAP] TX --> Configure Request (DCID: 0x0040)
[L2CAP] RX <-- Configure Request (DCID: 0x0041)
[L2CAP] TX --> Configure Response - Success (SCID: 0x0040)
E (656470) BT_BTM: btm_process_remote_ext_features: page=3 unexpected

[Crash] Crash detected at state TX / L2CAP / Configure Response
------------------------------------------------------------
[Optimizer] Iter=3  Params=[0.00472128,0.0751069,0.136429,0.190592,0.0687216,0.15383,...
[Optimizer] Fitness=47  Adj. Fitness=-47
------------------------------------------------------------
51:[LMP] TX --> LMP_detach
Host BDAddress randomized to 2f:57:93:6f:01:9c
[!] Global timeout started with 45 seconds
Guru Meditation Error: Core  0 panic'ed (IllegalInstruction). Exception was unhandled.

Core  0 register dump:
PC      : 0xdeadbeee  PS      : 0x00060031  A0      : 0x80087218  A1      : 0x3ffbe360
A2      : 0x3ffb80ec  A3      : 0x3ffb1a3c  A4      : 0x00000001  A5      : 0x000ffffc
A6      : 0xdeadbeee  A7      : 0x00060b23  A8      : 0x800832cc  A9      : 0x3ffbe340
A10     : 0x3ffb1a3c  A11     : 0x3ffba498  A12     : 0x00000000  A13     : 0x00000001
```

Figure 15: Output of Arbitrary code execution on ESP32 (CVE-2021-28138)

Following page 13 of the paper (DoS in Laptops & Smartphones), we can launch a denial-of-service attack against a smartphone (Oppo Reno 5G) and monitor it via ADB. To this end, change the parameter *"MonitorType"* to **3** in *configs/bt_config.json* (using nano or vim for example) and run the fuzzer with the *"invalid_timing_accuracy"* exploit.

```
 sudo bin/bt_fuzzer --no-gui --exploit=\
invalid_timing_accuracy --target=c0:2e:25:df:73:80 \
--target-port=/dev/ttyUSB3
```

Run the command above for about 2 minutes and stop the fuzzer with *CTRL + C*. Since the output of the phone via logcat is too fast, we need to manually check the target log (*logs/Bluetooth/monitor.1.txt*) to validate if a crash has been triggered on the SoC of the target.

```
cd $HOME/braktooth/wdissector/logs/Bluetooth
cat monitor.1.txt | grep -i "SoC Crashed"
```

If the target BT firmware has crashed and the attack was successful, the output of the command about should return the string ***"Primary Reason for SoC Crash:SOC crashed"***

Finally, the evaluator can optionally launch exploits to trigger the bugs in Table 2 of the paper. This customization is elaborated in Section A.7.4.

A.6.6 Experiment 6 - Fuzzing Extensions

This section evaluates the claim that our fuzzer is extensible to other wireless protocols such as Wi-Fi and BLE Host by running an exploit against ESP32 (BLE Host) and Raspberry Pi 3B (Wi-Fi). We leave the replicability of the "coverage" of Table 7 (Summary of unknown flaws found by extension) on the paper as optional since demonstrating the exploits confirms the extensibility of the fuzzer.

BLE Host Fuzzer: Starting with the BLE host fuzzer, we need to flash a BLE firmware to ESP32 with a slight modification to the sample code "gatt_security_server":

```
nano $HOME/esp-idf/examples/bluetooth/bluedroid/ble/\
     gatt_security_server/main/example_ble_sec_gatts_demo.\
     c
```

```
# Modify the following
-    .own_addr_type      = BLE_ADDR_TYPE_RANDOM,
+    .own_addr_type      = BLE_ADDR_TYPE_PUBLIC,

-            esp_ble_gap_config_local_privacy(true);
+            esp_ble_gap_config_local_privacy(false)
```

After modifying the source code as instructed above. You can build and flash the new firmware to the ESP32 target:

```
$HOME/esp-idf/
source export.sh
cd $HOME/esp-idf/examples/bluetooth/bluedroid/ble/\
    gatt_security_server/
idf.py build
idf.py -p /dev/ttyUSB3 flash_erase flash
```

Now, we can launch the *"Null Dereference"* exploit from the fuzzer as follows:

```
sudo bin/bthost_exploiter --target=08:3a:f2:31:1c:b2 \
    --exploit=esp32_bluedroid_pairing_crash
```

If the null pointer dereference attack (CVE-2022-26604) is successful, you should the output indicated in Figure 16.

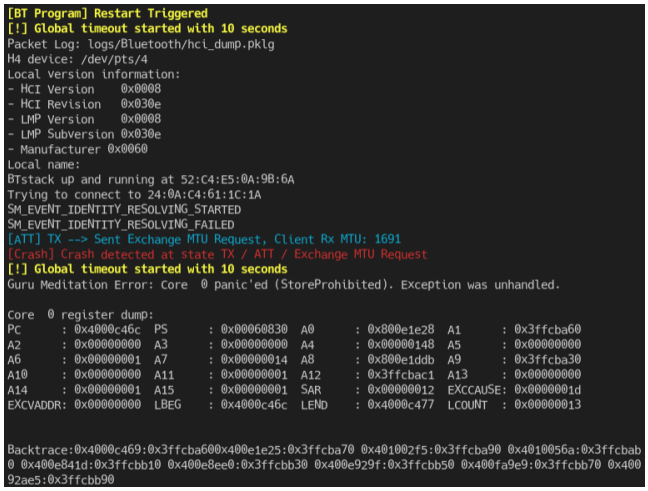


Figure 16: ESP32 Bluedroid Null Pointer dereference (CVE-2022-26604)

Moreover, to run the BLE Host in normal mode (without any exploits) and replicate Table 7, the fuzzer can be launched as follows:

```
sudo bin/bthost_exploiter --target=08:3a:f2:31:1c:b2 \
    --mutation=true
--duplication=true --optimization=true --max-iterations=1\
    000
cp logs/BTHost modules/eval/custom
cd modules/eval
./eval.py custom
```

The customization of this experiment to target other targets as shown in Table 2 is discussed in A.7.5.

Wi-Fi AP Fuzzer:

Next, we repeat the exploitation experiment for the Wi-Fi AP fuzzer by launching the *Probe Resp. Deadlock* (CVE-2022-26599) against Raspberry Pi 3B. To launch such attack, we need to force the Wi-Fi client (Raspberry Pi) to connect to our Wi-Fi AP by running a script that ensures reconnection Wi-Fi reconnection. We provide this Raspberry Pi script in our remote setup via SSH at 10.42.0.220:

Listing 6: Wi-Fi Client reconnection script

```
ssh pi@10.42.0.220 # No password needed
cd WiFiSuite/wifisuite/
sudo dmesg -C && sudo dmesg -w &
sudo python test.py
```

After the previous commands were issued in Raspbery Pi, it will try to connect to an AP matching the name (SSID) "TEST_KRA". Now, to start the Wi-FI AP fuzzer, start a new SSH terminal on the remote machine and run the following:

Listing 7: Wi-Fi Client reconnection script

```
cd $HOME/braktooth/wdissector
sudo bin/wifi_ap --exploit=broadcom_bad_prob_rsp
```

After a couple of minutes (about 1-2 minutes), the attack is successful if the Raspberry Pi SSH terminal shows the string *"firmware has halted or crashed"* (c.f., Figure 17).

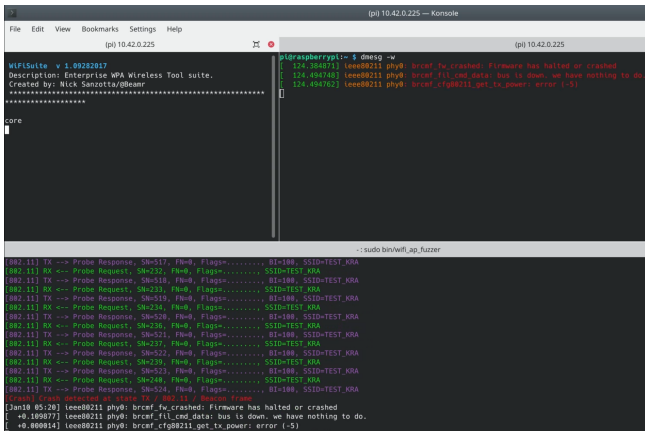


Figure 17: Raspberry Pi 3B Probe Response Deadlock (CVE-2022-26599)

Similarly to BLE Host, you can optionally run the Wi-Fi AP fuzzer in normal mode and replicate Table 7 as follows:

```
sudo bin/wifi_ap_fuzzer --mutation=true
--duplication=true --optimization=true --max-iterations=1\
    000
cp logs/wifi_ap modules/eval/custom
cd modules/eval
./eval.py custom
```

The customization of this experiment to target other devices as shown in Table 2 is discussed in A.7.5.

A.7 Experiment Customization

A.7.1 Experiment 1

To replicate all the results of Table 4 of the paper, you can launch the BT fuzzer with the following arguments:

```
sudo bin/bt_fuzzer --no-gui --target=<BDAddress> \
     --target-port=<Serialport> --duplication=true \
     --mutation=true --optimization=true
cp logs/Bluetooth modules/eval/custom
cd modules/eval
./eval.py custom
```

The arguments **–target** and **–target-port** corresponds to the columns *BDAddress* and *Monitor* of Table 1. Note that **–target-port** is only used if the target is using the *Serial* monitor type. For targets that use *SSH* or *ADB*, the file *configs/bt_config.json* need to be updated as follows:

- SSH:
  Update attribute *"MonitorType":1*
  Update attribute *"SSHUsername": "artifact"*
- ADB:
  Update attribute *"MonitorType":3*
  For Oppo Reno 5G, update attribute *"ADBDevice": "627ff0eb"*
  For OnePlus 5T, update attribute *"ADBDevice": "3ffd4d9a"*

After the changes above, you can start the fuzzer without argument *–target-port*.

Next, to replicate all the results of Table 5 of the paper with the correct number of Iterations and parameters (R_{sel} and D_T), the configuration file *configs/bt_config.json* can be changed as follows:
(*i*) "MaxIterations": 200
(*ii*) "DefaultDuplicationProbability": R_{sel}
(*iii*) "MaxDuplicationTime": D_T

A.7.2 Experiment 2

To generate your own BT captures to be used in state machine model generation (Experiment 2), you can disable the fuzzer components and increase the global timeout as follows:
(*i*) "enable_duplication": false
(*ii*) "enable_mutation": false
(*iii*) "enable_optimization": false
(*iv*) "GlobalTimeout": 9999 Then you can start the fuzzer without the argument *–mutation, –duplication, –optimization*. Example:

```
sudo bin/bt_fuzzer --no-gui --mutation=false
--duplication=false --optimization=false
```

Finally, after running the tool for a while, the reference wireshark captures are saved to *logs/Bluetooth/capture_bluetooth.pcapng*

A.7.3 Experiment 4

To run the tools against other BT devices from Table 1, use their respective *BDAddress* when starting each comparison script. For example, you need to manually modify script *experiment3_bss.sh:25* to use the correct *BDAddress* of the target device. For the other comparison scripts, you need to wait for the tools to scan the environment before selecting the correct *BDAddress* of the target BT device.

A.7.4 Experiment 5

In order to launch exploits to trigger the bugs reported in Table 2 of the paper (*Summary of unknown implementation bugs and other anomalies found*), we need the mapping between the Exploit Name and the vulnerability name as reported in Table 2 of the paper. Table 3 outlines the columns **Attack Name, Exploit Name**, CVE ID, Affected SoC, etc to capture this mapping.

An exploit can be launched from the fuzzer with the following arguments:

```
sudo bin/bt_fuzzer --no-gui --target=<BDAddress> \
     --target-port=<Monitor>
--exploit=<Exploit Name>
```

Note that the **<BDAddress>,<Monitor>** corresponds to Table 1 and **<Exploit Name>** corresponds to the last column of Table 3. Lastly, the argument *–target-port=<Monitor>* is optional and can be configured for ADB and SSH targets as discussed in Section A.7.1.

You can also list all the available exploits names that are stored in folder *modules/exploits/bluetooth/*.cpp* by running the following command:

```
sudo bin/bt_fuzzer --no-gui --list-exploits
```

Exploit creation and modification is out of the scope of this artifact, but a tutorial material is included in the documentation file *exploit_modules_tutorial.pdf* at the root folder of the artifact package.

A.7.5 Experiment 6

In order to evaluate the fuzzer against the BLE Targets of Table 2, you can launch the BTHost fuzzer as follows:

```
sudo bin/bthost_fuzzer --target=<Address> --target-port=<\
     Monitor> --duplication=true --mutation=true \
     --optimization=true
```

The parameter **Address** is informed by Table 2 and devices in which the column *Monitor* is *"N.A"* means that no monitor is applicable to such device. In this case, you can omit the argument *"–target-port"* before launching the fuzzer.

Similarly to the BTHost fuzzer, you can launch the Wi-Fi fuzzer as follows:

```
     sudo bin/wifi_ap_fuzzer --target-port=<Monitor> \
     --duplication=true --mutation=true --optimization=\
     true
```

Note that since the Wi-Fi fuzzer is a rogue AP which waits a connection from the target device, the column *Address* of Table 2 is not applicable.

Similar to the monitor configuration procedure of Section A.7.1, the argument **–target-port** is only used if the target is using the *Serial* monitor type. For targets that use *SSH* or *ADB*, the file *configs/wifi_ap_config.json* or *bthost_config.json* (depending on which fuzzer you are running) need to be updated as follows:

- SSH:
  Update attribute *"MonitorType":1*
  For Raspberry Pi, update attribute *"SSHUsername": "pi"*
  For Raspberry Pi, update attribute *"SSHHostAddress": "10.42.0.220"*
- ADB:
  Update attribute *"MonitorType":3*
  For OnePlus 5T, update attribute *"ADBDevice": "3ffd4d9a"*

After the changes above, you can start the fuzzer without argument *–target-port*.

Table 3: Summary of Exploits and Affected BT Devices

CVE ID	Attack Name	Affected Vendor(s)	Affected SoC(s) or Product(s)	Impact	Exploit Name
CVE-2021-28139	Feature Page Execution	Espressif Systems	ESP32 (SoC)	ACE / Deadlock	invalid_feature_page_execution
CVE-2021-28136	Duplicated IOCAP	Espressif Systems	ESP32 (SoC)	Crash (Reboot)	duplicated_iocap
CVE-2021-28135	Feature Res. Flooding	Espressif Systems	ESP32 (SoC)	Crash (Reboot)	feature_response_flooding
CVE-2021-28138	Invalid Public Key	Espressif Systems	ESP32 (SoC)	Crash (Reboot)	wrong_encapsulated_payload
CVE-2021-28137	Feature Req. Ping-Pong	Espressif Systems	ESP32 (SoC)	Crash (Reboot)	feature_req_ping_pong
CVE-2021-28155	Feature Res. Flooding	Harman International	JBL TUNE500BT (Product)	Crash (Shutdown)	feature_response_flooding
CVE-2021-31609	LMP Auto Rate Overflow	Silabs	WT32i (SoC)	Crash (Reboot)	lmp_auto_rate_overflow
CVE-2021-34147	Invalid Timing Accuracy	Infineon Technologies	CYW20735B1 (SoC)	Crash (Reboot)	invalid_timing_accuracy
CVE-2021-34146	AU Rand. Flooding	Infineon Technologies	CYW20735B1 (SoC)	Crash (Reboot)	au_rand_flooding
CVE-2021-34145	LMP Invalid Max Slot Type	Infineon Technologies	CYW20735B1 (SoC)	Crash (Reboot)	invalid_max_slot
CVE-2021-34148	LMP Max Slot Overflow	Infineon Technologies	CYW20735B1 (SoC)	Crash (Reboot)	lmp_max_slot_overflow
CVE-2021-34149	AU Rand. Flooding	Texas Instruments	CC2564C (SoC)	Deadlock	au_rand_flooding
CVE-2021-31610	AU Rand. Flooding	Bluetrum	BT889X / AB5XX / AB5301A (SoCs)	Crash (Reboot)	au_rand_flooding
CVE-2021-34150	LMP Length Overflow over DM1	Bluetrum	AB5301A (SoC)	Deadlock (Paging disabled)	lmp_overflow_dm1
CVE-2021-34143	AU Rand. Flooding	Zhuhai Jieli Technology	AC6366C (SoC)	Deadlock	au_rand_flooding
CVE-2021-34144	Truncated SCO Link Request	Zhuhai Jieli Technology	AC6366C (SoC)	Deadlock	truncated_sco_link_request
CVE-2021-31612	LMP Auto Rate Overflow	Zhuhai Jieli Technology	AC6905X (SoC)	Deadlock	lmp_auto_rate_overflow
CVE-2021-31613	Truncated LMP *accepted*	Zhuhai Jieli Technology	AC6905X / AC6925C (SoC)	Crash (Reboot)	truncated_lmp_accepted
CVE-2021-31611	Invalid Setup Complete	Zhuhai Jieli Technology	AC6905X / AC6925C (SoC)	Deadlock	invalid_setup_complete
CVE-2021-31787	Feature Res. Flooding	Actions Technology	ATS2815 / ATS2819 (SoC)	Crash (Shutdown)	feature_response_flooding
CVE-2021-31785	Repeated Host Connection	Actions Technology	ATS2815 / ATS2819 (SoC)	Deadlock	repeated_host_connection
CVE-2021-31786	Multiple Same Host Connection	Actions Technology	ATS2815 / ATS2819 (SoC)	Deadlock (Shutdown)	N.A (Specific BDAddress Configuration)
CVE-2021-33155	LMP Paging Scan Disable	Intel	Intel AX200 (SoC)	Deadlock (Paging disabled)	paging_scan_disable
CVE-2021-33139	Invalid Timing Accuracy	Intel	Intel AX200 (SoC)	Crash (FW Reboot)	invalid_timing_accuracy
CVE-2021-30348	Invalid Timing Accuracy	Qualcomm	Snapdragon 845 / 855 / Others (SoCs)	Crash (FW Reboot)	invalid_timing_accuracy
CVE-2021-35093	LMP Length Overflow over 2-DH1	Qualcomm	CSR 8811 / CSR 8510 (SoCs)	Deadlock / Crash	lmp_overflow_2dh1
Pending	LMP Invalid Transport	Beken	BK3266	Deadlock (Paging disabled)	lmp_invalid_transport
CVE-2019-9506	Knob (Extra - For testing only)	Many	Many	Entropy Reduction	knob

A.8 Notes

In case that the BT target ESP32 hangs the fuzzing process and does not seem to move forward, then you can reset ESP32 by running the following command:

```
cd $HOME/braktooth/wdissector/modules/eval
./flash_esp32.sh
```

Due to IP requirements with our Keysight partners, the main source code of the fuzzer is freely available only for academic research purposes upon request to https://src.braktooth.com. Students or Researchers with a valid university email, will receive an automated invitation to our Gitlab repository. Nevertheless, the source code of our ESP32 reverse engineering framework is available to public at https://github.com/Matheus-Garbelini/esp32_firmware_patching_framework.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

This artifact contains the source code of the AmpFuzz fuzzer as well as a number of scripts that were used to evaluate it on a number of programs from the Debian repositories. As the evaluation pipeline is configured to run in multiple docker containers, a linux-based host-system running the docker daemon is required. To confirm that the artifact is functional and to reproduce individual results, a commodity laptop or computer does suffice. E.g., development took place on a core i5 with 8GB of RAM, on which re-discovery of the `bosserver` amplification vulnerability takes less than 5 minutes. To re-run the entire pipeline, which fuzzes all targets five times for 24 hours, a system with a larger number of cores and more RAM is recommended (our experiments ran on a server with two Intel Xeon Gold 6230N and 512GB of RAM, on which the pipeline finished in about 4 days).

Lastly, this artifact also contains code to synthesize python code from identified amplification vulnerabilities, which can be used to develop amplification honeypots. This step only requires a working Python3 installation on the host system and can also be run on a commodity system.

Overall, this artifact should show that

- The AmpFuzz fuzzer is functional and *can* discover amplification vulnerabilities.
- The honeypot synthesis step is functional and produces python code.
- A full 5x 24h evaluation reproduces Table 2 (within the confidence intervals provided), similar maximum amplification factors to those shown in Figure 5, and similar results to those shown in Figure 4 for UDP-aware fuzzing.

A.2 Artifact check-list (meta-information)

- **Algorithm:** No new algorithm is presented.
- **Program:** No standardized benchmark is available. Instead, AmpFuzz is evaluated on 20 services from the Debian repositories.
- **Compilation:** AmpFuzz leverages LLVM11.0.1 and builds on the compile-time instrumentation from ParmeSan and Angora. All sources are included with the artifact and automatically built.
- **Transformations:** AmpFuzz requires no external program transformation tools.
- **Binary:** No binaries are required/included.
- **Model:** No model is used.
- **Data set:** The "evaluation data set" consists of 20 debian programs. A script to reproduce the dataset from public debian repositories is included.
- **Run-time environment:** AmpFuzz was tested on Linux, requires access to a running docker daemon and relies on bash, GNU make, and xargs.
- **Hardware:** No special hardware is requried (a x86_64 processer is assumed).
- **Run-time state:** The artifact is non-sensitive to run-time state.
- **Execution:** Other heavy loads on the system could impact results.
- **Security, privacy, and ethical concerns:** AmpFuzz only performs local testing of publicly available programs. Where possible, care has been taken to prevent fuzzed services from opening external network connections.
- **Metrics:** Included scripts and the AmpFuzz report on
  - Number of unique execution paths
  - Number of unique network requests (as defined by unique basicblock coverage)
  - Number of amplification inducing network requests
  - Maximum amplification factor on layer 2 (including Ethernet frames)
  - Maximum amplification factor on layer 7 (UDP payload only)
  - Elapsed seconds until first response
- **Output:** Each fuzz run produces
  - A human-readable console log with statistics (`fuzz.log`)
  - A csv file with fine-grained statistics (`angora.log`)
  - a folder with tested requests as individual files (`queue/id:<numeric_id>`)
  - a folder with amplification inducing requests as individual files (`amps/amp_<amp_factor_l2>_<path_hash>_<input_hash>`)

  Scripts are provided to generate the tables and figures included in the paper from these raw-results:
  - `02_print_table.py` produces LaTeX code on which Table 2 is based
  - `03_plot_grid.py` procudes Figures 4 and 5
- **Experiments:** Mostly automated, see detailed description below.
- **How much disk space required (approximately)?:** For verifying functionality on individual targets, 20GB should suffice (the source-code repository requires approximately 3GB, most of this from the LLVM git repository, the basic docker containers take around 13GB). Docker containers and output from the full evaluation fit on a 1TB drive.
- **How much time is needed to prepare workflow (approximately)?:** Building the initial docker containers (step 1 above) should take less than 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** Running the full evalution with no changed parameters (5 repetitions, 24 hour timeouts plus several 1 hour runs with different configurations) took about 4 days on a system with 80 threads.

- **Publicly available (explicitly provide evolving version reference)?:** AmpFuzz is publicly available at https://github.com/cispa/ampfuzz
- **Code licenses (if publicly available)?:** AmpFuzz is licensed under Apache License 2.0
- **Data licenses (if publicly available)?:** n/a
- **Workflow frameworks used?:** No workflow frameworks were used (evaluation pipeline only requires GNU make and xargs)
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/cispa/ampfuzz/releases/tag/usenix22_ae

A.3 Description

A.3.1 How to access

AmpFuz can be retrieved from GitHub via

```
git clone --recursive
↪ https://github.com/cispa/ampfuzz
```

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

- Linux host OS
- Docker
- bash
- GNU make
- GNU xargs
- Python3 with packages `pandas`, `numpy`, `seaborn`

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

AmpFuzz only performs local testing of publicly available programs. Where possible, care has been taken to prevent fuzzed services from opening external network connections.

A.4 Installation

A.4.1 Build docker base images

from the project directory, run

```
make
```

This will take some time and build four docker images:

- `ampfuzz:base`: serves as the base-image for the other three stages, basically a Ubuntu 20.10 image with some packages installed and including a copy of the llvm source.
- `ampfuzz:wllvm_wrapper`: used to build ubuntu packages with `wllvm`, a the whole-program LLVM wrapper. Our later stages use `wllvm` to extract LLVM bitcode from installed packages.
- `ampfuzz:fuzzer`: includes the fuzzer and required instrumentation tools.
- `ampfuzz:symbolic_execution`: includes the symcc symbolic execution engine, and is used to instrument targets and replay the amplification inputs to collect path constraints.

A.5 Evaluation and expected results

We claim that

1. AmpFuzz fuzzer is functional and can discover amplification vulnerabilities (Table 2 in the paper)
2. UDP-aware fuzzing allows AmpFuzz to find amplification inducing responses *faster* than static timeouts (Figure 4 in the paper)
3. The amplification maximization routines of AmpFuzz can lead to higher maximum amplification factors than coverage-guided fuzzing alone (Figure 5 in the paper)

A full evaluation run should produce results from which Table 2, Figure 4 and Figure 5 could be reproduced (within the confidence intervals provided in the paper).

A.5.1 Prepare Evaluation Directory

from the `eval` subdirectory, run

```
make
```

This will generate a fresh evaluation directory in `eval/04_create_eval_dir/eval`. This directory contains

- **args**: A textfile containing the different fuzzer configurations and timeouts
- **build_scripts**: helper scripts to build containers used for fuzzing
- **eval_scripts**: helper scripts to analyze results (see below)
- **fuzz_all.sh**: bash script to run entire fuzzing pipeline
- **fuzz_scripts**: helper scripts used during fuzzing
- **hpsynth_scripts**: helper scripts used during honeypot code synthesis
- **Makefile**: a GNU make script with rules to build containers used for fuzzing (makes use of `build_scripts`)
- **targets**: configuration info for fuzz targets.
  - **<debian_package>/<path_to_binary_escape>/<port>**: configuration directory for a single fuzz target. Will also be used to store log-files and container information.
    - **args**: commandline arguments to be passed to the fuzz target

    - **config.sh**: bash script that configures the fuzz target for fuzzing
- **targets.json**: json file containing tuples of
    1. debian package
    2. path to binary
    3. port number

  for all fuzz targets.

This newly built `eval` directory can be moved around freely. Everything from here onwards will happen within this directory!

A.5.2 Fuzz

Running `fuzz_all.sh` within this newly created eval directory will now

1. use the generated `Makefile` to prepare all targets for fuzzing (i.e., building and instrumenting the target into individual docker images)
2. fuzz each target with each configuration and collect all results into a new `results` directory
3. run the paths-to-message deduplication script. This script collects all unique "paths" found during fuzzing and executes them against the dataflow-instrumented target binary, collecting only request-dependent CFG edges.

For each target and run, a new subfolder will be created of the form `results/<pkg>/<binary>_<port>/<run>`.

A.5.3 Analyze results

Once fuzzing and path-deduplication has completed, the new `results` directory can be analyzed:

1. `eval_scripts/01_compute_amp_stats.py` will extract final stats for each run into a file `results/results.json`
2. `eval_scripts/02_print_table.py` will generate latex code for the overview table shown in the paper
3. `eval_scripts/03_plot_grid.py` will generate the plots to show the results of different timeouts and amplification maximization runs

A.5.4 (optional) generate honeypot code

Prepare a target for symbolic exeuction, run constraint-collection for a run folder (`results/<pkg>/<binary>_<port>/<config>/<run>`) and convert the collected constraints to python code:

1. Build a docker container for symbolic execution of the target:

   ```
   make targets/<debian_package>/.sym_config_<path
   ↪ _to_binary_escaped>_<port>.iid
   ```

2. `bash hpsynth_scripts/synth_one.sh <run_folder>` will create a constraints file named `hpsynth/sym.result` in the run folder.
3. `python hpsynth_scripts/main.py <sym.result>` will output python code for a number of `check` and `output` functions, along with a combined `gen_reply` function.

(Honeypot-skeleton for listening on ports and providing rate-limiting is not provided with this project)

A.6 Experiment customization

A.6.1 Full pipeline customization

Evaluation is controlled by two files, `args` and `fuzz_all.sh`. `args` contains the different fuzzing configurations, one per line, in the following format

```
<output_directory> <timeout> [extra_args ...]
```

E.g., the two lines

```
24h 24h
1h_100ms 1h -a=--disable_listen_ready
↪ -a=--early_termination=none
↪ -a=--startup_time_limit=100000
↪ -a=--response_time_limit=100000
```

will run

1. a default configuration for 24 hours and store the results into directory `24h`
2. a configuration with a static timeout of 100ms and store the results into directory `1h_100ms`

The `fuzz_all.sh` script further specifies how often each experiment should be repeated. This is controlled with the `N_RUNS` variable (defaults to `5`).

A.6.2 Individual results

Individual targets (e.g., for confirming functionality) can be tested as follows: First, to build a docker container for a specific target, run

```
make targets/<debian_package>/.fuzz_config_<path_to
↪ _binary_escaped>_<port>.iid
```

to build one of the configred targets. For example, to prepare a container to fuzz `/usr/sbin/bosserver` from the package `openafs-fileserver` on port 7007, use

```
make targets/openafs-fileserver/.fuzz_config__usr_s
↪ bin_bosserver_7007.iid
```

This will

- download the package sources for the target from the debian repositories
- compile it using wllvm
- install the package
- instrument the target binary for fuzzing
- apply additional configurations (from files `args` and `config.sh` found under `targets/<debian_package>/<binary>/<port>`)

A list of all valid targets can be retrieved using

```
make -qp|grep -oE
↪ 'targets/[^/]*/.fuzz_config[^/]*iid'|sort -u
```

To fuzz the target in its newly built container, run

```
bash fuzz_scripts/fuzz_one -r <runid>
↪ <debian_package> <path_to_binary>
↪ <port> <result_directory> <timeout>
```

where <runid> is just some number to identify the run (used as part of the output path).

Sticking with the example, to fuzz /usr/sbin/bosserver for 5 minutes and storing the results under results/openafs-fileserver/_usr_sbin_bosserver_7007/5m/01/, use

```
bash fuzz_scripts/fuzz_one.sh -r 1
 ↪ openafs-fileserver /usr/sbin/bosserver 7007
 ↪ results/openafs-fileserver/_usr_sbin_bosserver_
 ↪ 7007/5m
 ↪ 5m
```

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

Today's voice personal assistant (VPA) services have been largely expanded by allowing third-party developers to build voice-apps and publish them to marketplaces (*e.g.*, the Amazon Alexa and Google Assistant platforms). In an effort to thwart unscrupulous developers, VPA platform providers have specified a set of policy requirements to be adhered to by third-party developers, *e.g.*, personal data collection is not allowed for kid-directed voice-apps. In this work, we aim to identify policy-violating voice-apps in current VPA platforms through a comprehensive dynamic analysis of voice-apps. To this end, we design and develop SKILLDETECTIVE, an interactive testing tool capable of exploring voice-apps' behaviors and identifying possible policy violations in an automated manner. Distinctive from prior works, SKILLDETECTIVE evaluates voice-apps' conformity to 52 different policy requirements in a broader context from multiple sources including textual, image and audio files. With SKILLDETECTIVE, we tested 54,055 Amazon Alexa skills and 5,583 Google Assistant actions, and collected 518,385 textual outputs, approximately 2,070 unique audio files and 31,100 unique images from voice-app interactions. We identified 6,079 skills and 175 actions potentially violating at least one policy requirement.

A.2 Artifact check-list (meta-information)

- **Algorithm: This work does propose an algorithm for a question-type classifier.**
- **Program: The program components do require many different dependencies which are provided in the repository or listed and easily downloaded.**
- **Data set: All applicable data sets are included in the repository.**
- **Run-time environment:**
- **Hardware:**
- **Metrics: VPA device output data are gathered and after analysis, any potential policy violations should be reported**
- **Output: The outputs consist of VPA device interaction data saved in a data set, collected device output image and audio files. Lastly, after analysis, any potential policy violations should be reported.**
- **Experiments: There are detailed instructions provided for installation and software use. To run the experiment, one would only have to run the software after installation.**
- **How much disk space required (approximately)?: Approximately 300 to 400 GB.**
- **How much time is needed to prepare workflow (approximately)?: It should take no longer than 30 minutes to set up the software.**
- **How much time is needed to prepare workflow (approximately)?: It should take no longer than 30 minutes to set up the software.**
- **Publicly available?: The artifact is available at https://github.com/skilldetective/skilldetective/releases/tag/V0.3**
- **Archived (provide DOI)?: The artifact is archived at https://github.com/skilldetective/**

A.3 Description

A.3.1 How to access

All of the software can be found at https://github.com/skilldetective/

A.3.2 Software dependencies

There is a list of software dependencies provided within the repository. Also, all of the dependency software for the Java components are included in the repository and all of the needed Python dependencies can be easily downloaded and are clearly stated in the instructions documents.

A.4 Installation

Installation of SD requires a java IDE such as NetBeans and a version of Python installed. The chatbot model runs on Java and has a detailed installation guide that walks the user through all the steps necessary such as acquiring a developer account, writing a test app, accessing the testing terminal, installing and running the software and what to expect from the output as well as some troubleshooting. The policy compliance portion of the software package has a detailed installation and user's guide that outlines all needed steps to analyze the outputs of the chatbot.

A.5 Experiment workflow

The chatbot should be installed and run first. We have provided a list of Alexa skill names for testing. Device interactions should be collected automatically for at least 30 minutes to an hour in order to insure an adequate amount of test data get collected. Next, the policy compliance software should be installed and the test data from the chatbot can be used for evaluation.

A.6 Evaluation and expected results

The chatbot runs autonomously once installed and set up. A data set consisting of speech interactions, image files and audio files should be expected as output. These output data can then be analyzed using the policy compliance software. As a final output, the user should expect a list of any suspected policy violations found in the interaction data.

A.7 Experiment customization

The experiment has many different changeable parameters. First, the list of application names can be altered to include any arbitrary set by changing the file Skills.xlsx. The web browser used to collect the interaction data can be changed by altering the selenium code in S5.java. There are a number of software parameters such as how many interactions are allowed per VPA application and possible changes made to the neural network. These are all commented within the source code.

A.8 Notes

Please feel free to contact us at anytime with any concerns or issues. The email address is skilldetectivetroubleshoot@gmail.com. Also, within the repository please make sure to follow the instructions located at:

- https://github.com/skilldetective/skilldetective/tree/master/skilldetective_policy_detector
- https://github.com/skilldetective/skilldetective/blob/master/ChatBot/SkillDetective%20Instructions.pdf



A Artifact Appendix

A.1 Abstract

We make the artifacts of our large scale geographical study on the geodifferences in mobile apps available to the research community. In this research, we crawl a set of 5,684 globally popular app binaries, their metadata, and privacy policies from Google Play store in 26 countries. We open-sourced the crawlers we used to download these apps, their metadata, and privacy policies, and provide documentation on how to setup these crawlers in a public *GitHub* repository. In addition, we provide a way for researchers to request access to the actual app binaries and privacy policies. The metadata of these apps and the error messages that we obtain during our crawls are both published and available for download.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** The code requires a Python 3 environment. The repository contains three crawlers (for metadata, APK, and privacy policies) and it is recommended to create separate environments for each crawler; we have provided Pipfiles to install Python dependencies to each environment. Instructions are located in the repository READMEs.
- **Output:** Depending on which code is run, the crawler will download the respective data (metadata, APK, privacy policy) to the output folder specified by the user. Note that the crawler is making requests to Google Play (or sites hosting individual privacy policies) and may have nondeterministic output depending on the time of the request.
- **Publicly available:** Our code is publicly available on GitHub and we provide a public Google Drive URL (link in the repository README) to download our app metadata and crawl error messages. The privacy policy and APK files are available upon request.
- **Code licenses:** The code is released under the GNU General Public License. Full licensing information is provided in the GitHub repository.
- **Archived:** A stable link to our GitHub repository can be found at the following link: `https://github.com/censoredplanet/geodiff-app/tree/9ae97196ee82e741e17126dfc6ad518a88ea2cac`. We have a public Google Drive folder containing a subset of our dataset for download: `https://drive.google.com/drive/folders/1-UGiOUEEge-DA53k9B7KbIOvMlXKfiYZ`.

A.3 Description

A.3.1 How to access

The complete source code for the crawlers we customized for this research along with their setup instructions is available as a public *GitHub* repository here `https://github.com/censoredplanet/geodiff-app` (commit hash *9ae97196ee82e741e17126dfc6ad518a88ea2cac*). The app binaries and their privacy policies are available upon request (write to us at geodiff.app@umich.edu and we will provide a download link). We have provisioned it this way for us to keep track of who has access to our data at a time. The metadata for the apps, and Google's error messages we obtained at the time of app downloads (which we used to determine who is resposbile for blocking) are available at `https://drive.google.com/drive/folders/1-UGiOUEEge-DA53k9B7KbIOvMlXKfiYZ`.

A.3.2 Data sets

Refer to subsubsection A.3.1 on how to access our dataset.

A.4 Installation

Our crawlers use the runtime environment provided by the `Python` interpreter. We use `Pipfiles` to manage `Python` dependencies. The installation instructions are provided as a `README` on our GitHub repository accessible at the URL: `https://github.com/censoredplanet/geodiff-app/tree/9ae97196ee82e741e17126dfc6ad518a88ea2cac`.

A.5 Evaluation and expected results

Our setup involved ten Linux machines, ten Android phones, and VPN/S access to 26 countries. Once the measurement testbed is setup (as described in the paper), the crawlers can be used to download the apps from each respective country. However, note that reproducing the entire setup and experiment results depends on several factors, including the availability of reliable VPN/S access for downloads and the apps we tested on Google Play. That apart, the results may not be precisely reproducible given how these apps are updated regularly.





A Artifact Appendix

A.1 Abstract

Our artifacts facilitate building and running Morphuzz for the QEMU and Bhyve hypervisors. These are the two implementations of Morphuzz described in our paper. We packaged the fuzzers in two VMs (one for fuzzing each hypervisor). Use an Intel/AMD x86-64 Linux machine to run these VMs.

A.2 Artifact check-list (meta-information)

- **Compilation:** clang (included)
- **Run-time environment:** Linux
- **Hardware:** Intel/AMD x86-64 Machine
- **Output:** Crashes
- **How much disk space required (approximately)?:** 20 GB
- **How much time is needed to prepare workflow (approximately)?:** 1-2 Hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GPLv2
- **Archived?: https://zenodo.org/record/5655839**

A.3 Description

A.3.1 How to access

Download the Artifacts (2 VM images) from: https://zenodo.org/record/5655839

A.3.2 Hardware dependencies

Intel or AMD x86-64 Machine with Virtualization support

A.3.3 Software dependencies

Linux, KVM, and QEMU 3.1+

A.4 QEMU Installation

Our instructions are based around VM images. The evaluation should be possible on a single bare-metal x86 machine running linux.

Please install QEMU to run these VMs:

- Debian: `apt install qemu-system-x86_64`
- Ubuntu: `apt install qemu-system-x86_64`
- Fedora: `yum install qemu-kvm`
- Other: https://www.qemu.org/download/

Additionally, to ensure that the VM takes advantage of hardware-acceleration

1. Check that you have virtualization extensions enabled in BIOS (VT-x on Intel, AMD-V/SEV on AMD).
2. Check that your user has access to `/dev/kvm`. Usually, you can add your user to the kvm group: `sudo adduser $USER kvm`. (Otherwise you may need to run qemu as root/with sudo).
3. To test out bhyve-fuzzing, you may need to enable support for nested-virtualization. Unlike the QEMU-Fuzzer, the bhyve-Fuzzer is not decoupled from the in-kernel CPU virtualization component. Thus, even though our fuzzers do not run any virtual CPU code, we will need to enable nested-virt to create the VM. This amounts to loading the KVM kernel module with a special flag. This page has instructions to do this for AMD and Intel CPUs: https://docs.fedoraproject.org/en-US/quick-docs/using-nested-virtualization-in-kvm/

Please open a terminal and "cd" into the folder containing the qcow2 VM images you downloaded. We will be building the fuzzers inside the VMs. Then, we will fuzz a virtual-device. We will observe any crashes found by the fuzzer. Finally, we will generate coverage-reports for the results.

In each VM, we provide annotated scripts that will build the fuzzer and run it for some example virtual-device.

Boot the QEMU Fuzzing VM:

```
$ qemu-system-x86_64 -machine q35 \
-accel kvm -cpu host -m 4G -smp 2 \
-hda ./morphuzz_qemu.qcow2 -vga virtio \
-device virtio-net,netdev=mynet0 \
-netdev \
user,id=mynet0,hostfwd=tcp:127.0.0.1:22222-:22
```

After a few seconds, you should be able to ssh into the VM from another terminal on your machine:

```
$ ssh -p22222 paper@localhost
Credentials:
    user: paper
    pass: artifact_eval
```

Once you are SSHed, we can proceed with building and running the fuzzer.

A.5 QEMU Evaluation

We provide annotated scripts for building, fuzzing, and providing readable qtest-reproducers:

```
$ cat build.sh        # Examine the build script...
$ ./build.sh          # Build QEMU with Morphuzz

$ cat run_example.sh  # Examine the example script
                      # for running the fuzzer
$ ./run_example.sh    # Fuzz a virtual device
# ctrl-c to stop fuzzing

$ cat reproducer.sh   # Examine the script
                      # to build a QEMU reproducer

$ ./reproduce.sh      # This will reproduce
                      # a megaraid bug

$ ./build_gcov.sh     # Build Morphuzz with
                      # GCov Support

$ ./run_gcov.sh       # Run the CORPUS collected by
                      # run_example.sh and output a
                      # coverage summary
```

Once you are ready to switch to the bhyve VM: `sudo shutdown`

A.6 bhyve Installation

Boot the bhyve Fuzzing VM:

```
$ qemu-system-x86_64 -machine q35 \
-accel kvm -cpu host -m 4G -smp 2 \
-hda ./morphuzz_bhyve.qcow2 -vga virtio \
-device virtio-net,netdev=mynet0 \
-netdev \
user,id=mynet0,hostfwd=tcp:127.0.0.1:22223-:22
```

After a few seconds, you should be able to ssh into the VM from another terminal on your machine:

```
$ ssh -p22223 paper@localhost
Credentials:
    user: paper
    pass: artifact_eval
```

A.7 bhyve Evaluation

```
$ cat build.sh          # Examine the build script...
$ sudo ./build.sh       # Build Bhyve with Morphuzz

$ cat run_example.sh    # Examine the example run script
$ sudo ./run_example.sh # Fuzz Bhyve configured with
                        # common virtual-devices
# ctrl-c to stop fuzzing

$ sudo ./reproduce_crashes.sh   # This will reproduce
                                # the crashes discovered
                                # by Morphuzz

$ sudo ./run_cov.sh     # This will output a fuzzing
                        # coverage report to /tmp/html

# Use scp to copy the coverage report to the local
# machine. View the report in a web browser.
```

Note that some of these commands require sudo. Once you are done, `sudo poweroff`

A.8 Experiment customization

The QEMU/Bhyve configurations can be customized by changing the environment variables specified in the `run_example.sh` script. These variables specify the virtual devices attached to QEMU/Bhyve.

A.9 Notes

These steps are specific to the artifacts provided in the VMs. An up-to-date version of QMorphuzz is maintained and documented at https://gitlab.com/qemu-project/qemu/[1]

Current upstream documentation for using QEMU's fuzzing infrastructure/QMorphuzz can be found at: https://gitlab.com/qemu-project/qemu/-/blob/c39deb218178d1fb814dd2138ceff4b541a03d85/docs/devel/fuzzing.rst

The main differences between the upstream version of Morphuzz, and the version described in this paper are:

- The upstream version of QMorphuzz performs PCI enumeration, prior to fuzzing, to improve fuzzing efficiency.
- The upstream version contains some device-specific fuzzers (independent of QMorphuzz), which serve mostly as examples to go along with documentation. These are removed in the artifact.
- The upstream version of QMorphuzz provides a sparse memory device which improves the efficiency of the fuzzing process.
- The upstream version of QMorphuzz includes the configurations used to fuzz QEMU on OSS-Fuzz.
- The version of QEMU in the artifacts is 5.0.0. At this time, QEMU 6.2.0 has been released. Many of the bugs that can be found by Morphuzz in the artifact VM have already been patched.
- The upstream version comes with documentation for fuzzing additional devices, and adding custom QEMU fuzzers.

[1] Stable link to the version of QEMU at the time of this writing: https://gitlab.com/qemu-project/qemu/-/tree/c39deb218178d1fb814dd2138ceff4b541a03d85



A Artifact Appendix

A.1 Abstract

FUZZWARE is a firmware emulation and fuzzing prototype which makes use of symbolic execution to model MMIO accesses. In our experiments, we fuzz tested different sets of samples (synthetic, state-of-the-art, and new targets for CVE discovery). Based on the experiment stage (analogous to our paper's evaluation subsections), we collect additional data such as modeling statistics, job timings, unit test coverage, and code coverage.

As a fuzzing work, our experiments require computational resources. At a minimum (for a single-iteration evaluation of our core experiments instead of the 5/10 iterations that we performed), you should expect to perform 42 CPU days worth of fuzzing time on a single Linux system during the evaluation period (21 for the state-of-the-art comparison only). For the easiest (and repeated) reproduction, we recommend 41 dual-core Ubuntu cloud VMs (2 CPUs / 4-6GB RAM / 64 GB storage) which will run for 11 days to perform the full replication. Other setups are possible, but will require a bit of tinkering with experiment run scripts.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Locally-scoped Dynamic Symbolic Execution
- **Program:** Fuzzware builds on top of AFL/AFL++, unicorn engine, angr (8.19.10.30), Python < 3.10 (due to angr version)
- **Compilation:** clang
- **Binary:** Firmware samples used for evaluation
- **Data Set:** Included: P2IM Unit Tests, P2IM Targets, uEmu Targets, Artificial Password Firmware Samples, Contiki-NG & Zephyr-OS Target Samples
- **Run-time environment:** Linux, Docker
- **Hardware:** The recommended setup for full replication requires 41 dual-core Ubuntu cloud VMs.
- **Metrics:** Unit Test Coverage, Model Generation Timings, MMIO Overhead Elimination Statistics, Code Coverage, Reached Bugs
- **Output:** Included: Crashes (binary files), Generated: Fuzzing Inputs (binary files), MMIO Models (text files), statistics (text files), GNU plots (PNG files)
- **Experiments:** bash scripts, readmes
- **How much disk space required (approximately)?:** Recommended setup: 25 GB of local storage for collected experiment results, and 41 Ubuntu cloud machines with 64GB storage each. For a fully local setup (run script customizations are required), 100GB should suffice to hold all experiment data.
- **How much time is needed to prepare workflow (approximately)?:** 4h
- **How much time is needed to complete experiments (approximately)?:** 5-10 days (on 41 Ubuntu cloud machines, total CPU time: 320 days for full experiment repetition count, 42 days for a single iteration)
- **Publicly available (explicitly provide evolving version reference)?:** Evolving: https://github.com/fuzzware-fuzzer
- **Publicly available?** Yes. Stable version: sec22-ae-accepted
- **Code licenses (if publicly available)?:** Apache-2.0
- **Data licenses (if publicly available)?:** Apache-2.0
- **Archived (explicitly provide DOI or stable reference)?:** 10.5281/zenodo.6499215

A.3 Description

A.3.1 How to access

We release both the research prototype, as well as all experiments and data as open source on GitHub at the following two locations:

- https://github.com/fuzzware-fuzzer/fuzzware/tree/sec22-ae-accepted
- https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/sec22-ae-accepted

A.3.2 Hardware dependencies

For the experiment reproduction, x86 computation resources are required. For the easiest reproduction (no customization of run scripts), 41 dual-core Ubuntu cloud instances are recommended (2 cores, 4-6GB RAM, 64GB storage). With run script customizations, other hardware setups that allow for 320 days worth of fuzzing computation time within a reasonable time frame can be used. In case a single-run reproduction is deemed sufficient, a total of 42 days worth of computation time (plus some compute for trace generation and metric aggregation) are required.

A.3.3 Software dependencies

We recommend Linux/Docker as the experiment platform for a reproducible setup of all dependencies. For a seemless reproduction, we further recommend an Ubuntu LTS release (e.g., 18.04 or 20.04). Note that the version of angr which is pinned for the evaluation constrains the python version to be <3.10, which means that Ubuntu LTS releases of coming years may require installing an older version of python than are the default for future Ubuntu releases.

A.3.4 Data sets

We include all required firmware samples for running the experiments in the GitHub repository. In more detail, we include a list of target firmware images from previous work (P2IM, uEmu), and compiled additional targets for bug discovery, which are present as pre-built binaries in the fuzzware-experiments repository.

To reproduce our newly introduced target firmware samples, we further provide build scripts for all relevant targets.

A.3.5 Models

We do not include the MMIO models generated by Fuzzware. These will be generated by the prototype on-the-fly during the experiments.

A.3.6 Security, privacy, and ethical concerns

The fuzzware-experiments repository contains crash cases for security critical bugs in ZephyrOS and Contiki-NG. All corresponding vulnerabilities have been disclosed to the maintainers and patches were developed.

A.4 Installation

Installing individual instances of Fuzzware is done via the build_docker.sh script (for a Docker-based setup), and via the install_local.sh script (for a native setup), which are both located in the fuzzware repository.

To ease the setup process for the experiments, we also created scripts in the fuzzware-experiments repository. to remotely install SSH-accessible Ubuntu instances. The corresponding script can be found in ssh_hosts_install.py. The README files within the repository and comments in the source code are meant to provide additional information to allow for the use of scripts and the customization of the installation process.

The recommended setup is to create 41 Ubuntu LTS hosts with 2 cores, 6GB RAM, and 64 GB of disk space each. This could be reduced 4 GB RAM and 32 GB disk space in case costs require minimizing.

A.5 Experiment workflow

The data required to conduct each experiment is contained within the fuzzware-experiments repository. The repository is organized in a way such that each subdirectory corresponds to a particular section in the paper. The mapping from directory to paper can be found in the top-level README.md file.

In essence, each experiment entails a 24-hour fuzzing run of the target (invoked via the "fuzzware pipeline" utility), which creates a *fuzzware-project* directory. This directory contains the state of the MMIO model configurations, as well as inputs that were generated by fuzzers over the span of the fuzzing run. In a second step, metrics such as code coverage are aggregated from this raw data. Finally, in a third step, depending on the experiment, additional aggregation is performed over the full set of fuzzing runs belonging to the given experiment. This aggregation collects a summary of the data as can be found in the respective section of the paper. Below we describe the workflow for each of these experiments.

(1) PW discovery & Unit Tests. For the first experiment, the following workflow can be used to reproduce the experiments:

1. Make sure to have installed fuzzware on cloud hosts via ssh_hosts_install.py. If 41 dual-core hosts have been installed with the expected naming conventions, no modifications to run scripts should be required.
2. Navigate to 01-access-modeling-for-fuzzing/pw-discovery/
3. Run the experiments on the hosts by executing the ssh_based_kickoff_experiments.sh script
4. The experiments are spawning tmux sessions on the remote machines, so *tmux list-sessions* should provide a status on running experiments.
5. Collect the results from the fuzzing runs after the experiments have been finished (10 repetitions of 24 hour runs). This is done via the ssh_based_collect_results.sh script. The fuzzers shut themselves down automatically, so the experiment does not have to be cancelled manually. You may run the script at any time to collect intermediate results, but for the final result it is best to wait until the fuzzer has shut itself down. You may check whether the experiment is still running by checking the corresponding tmux session. Expect the experiments to run for 10-11 days including trace and per-run metrics generation.
6. Compute summary metrics via the run_metric_aggregation.py script.

For the P2IM unit tests of experiment (1), you may run 01-access-modeling-for-fuzzing/p2im-unittests/run_experiment.sh and observe the stdout output.

(2) State-of-the-art comparison. For the second experiment, the following workflow can be used to reproduce the experiments:

1. Same as for experiment (1).
2. Navigate to 02-comparison-with-state-of-the-art
3. Same as for experiment (1).
4. Same as for experiment (1).
5. Same as for experiment (1), but with 5 repetitions, 5-6 days of runtime, and using ssh_based_collect_results.sh.
6. Same as for experiment (1).

Note that experiments (1) and (2) are meant to be run in parallel in case 41 hosts are present, as experiment (1) is pre-configured to use 20 instances, while experiment (2) is pre-configured to use the remaining 21 instances.

(3) CVE discovery. For the third experiment, we tested the targets in large-scale fuzzing campaigns. As such, single 24-hour runs for replication do not make sense in this context. Instead, we provide crashing proof-of-concept inputs which were all generated in fuzzing runs, alongside with README's giving context on each POC. An example of this is 03-fuzzing-new-targets/zephyr-os/prebuilt_samples/CVE-2020-10065/POC/ within the fuzzware-experiments repository. You can still run the different CVE targets using the fuzzware pipeline utility (please refer to *fuzzware pipeline -h* for more documentation). We built each target such that previously known bugs are fixed (e.g., bugs of related CVEs), and crashing inputs generated via fuzzing should have a high likelihood to trigger the CVE bug.

(4) Crash Analysis. For the fourth experiment, we provide crashing POC inputs alongside some documentation on each input. The experiment's README at 04-crash-analysis/README.md contains an overview of how POC inputs correspond to the previous experiments, and how they can be run in Fuzzware.

A note on the multi-host setups. The base setup for (1) and (2) expect that the experiments are run in a distributed fashion on multiple hosts. In case your hardware resources do not allow for this multi-host setup, it is also possible to perform the same experiments on a smaller number of hosts that have access to more CPU cores. We provide scripts to run the experiments locally in the form of *run_experiment.sh* scripts within the respective experiment directories. As we cannot predict the exact computation resources (one very large host, a handful of medium-sized hosts, ...), these scripts are configured to run without parallelization by default. This means without modification, simply running the different *run_experiment.sh* scripts will take nearly a year to complete. However, we built parallelization and target specification options via environment variables into these scripts, such that the *run_experiments.sh* scripts should aid you in

running the experiments according to a given hardware environment. Please refer to the script documentation for information on how to parallelize running the experiments within a host.

A.6 Evaluation and expected results

The main claims of the paper are:

1. Fuzzware employs a lightweight MMIO modeling technique.
2. Fuzzware's MMIO models reduce the fuzzer's input space considerably.
3. Fuzzware's MMIO models are applicable to a wide variety of firmware and hardware platforms.
4. Fuzzware outperforms the state-of-the-art.
5. Fuzzware's is able to identify previously unknown bugs.

The key results reported in the evaluation of the paper which support our claims are as follows:

1. Fuzzware's model generation cost an average of 6.34 minutes over 24-hour runs (6 seconds per model) for the pw-discovery data set.
2. On the same data set, Fuzzware achieves a minimum input elimination of 49.3% and a maximum 83.4%.
3. Fuzzware passes all of the valid P2IM unit tests.
4. In terms of basic block coverage, Fuzzware achieves on average 44% more coverage compared to P2IM and 61% more compared to uEmu.
5. Fuzzware's fuzzing campaigns yielded multiple previously unknown bugs in Zephyr-OS and Contiki-NG.
6. The majority of crashing inputs found by Fuzzware are true positives.

These claims are supported by the data generated when following the experiments in Section A.5. Note that the README file in each experiment sub-directory should provide additional context on what data is collected, where to find it, and what the expected results are.

After running the experiments, you should have access to a set of fuzzware-project directories that contain aggregated data. As an example, for the ARCH_PRO target of experiment (1), a directory *fuzzware-project-run-01* inside 01-access-modeling-for-fuzzing/pw-discovery/ARCH_PRO/ should have been automatically created. Similarly, for the P2IM/PLC sample of experiment (2), *fuzzware-project-run-01* should be present in 02-comparison-with-state-of-the-art/P2IM/PLC/. Running the *run_metric_aggregation.py* scripts should now output data in a similar representation to what can be found in the paper with regards to claim 1–4[1].

For claim 5–6, you may replay the given POC inputs and verify emulation behavior. In case you fuzz-tested the CVE targets with sufficient computation resources, you can also manually analyze the crashes which are produced in the respective fuzzware-project directories. We further include empiric timings for the first occurrence of according crashes in our experiments in 03-fuzzing-new-targets/README.md, alongside numbers on how many cores we used for the crash reproduction. Information on the reported bugs can be found in 03-fuzzing-new-targets/bug-details.

A.7 Experiment customization

In case your computation resources differ from our recommended setup, then modifications to the run scripts may be required to achieve experiment parallelization which matches your available setup. Please refer to the scripts' sources and README's for more information.

A.8 Notes

Due to the probabilistic nature of fuzzing, many of the numbers will differ in each run.

Furthermore, our basic block coverage collection is slightly different from the way it is collected in original publications for uEmu and P2IM. These papers report QEMUs translated blocks as reached basic blocks. However, due to the intrinsics of this emulator, these do not necessarily correspond to actual basic blocks. In our experiment, we match the entry of translated blocks to a list of actual basic blocks. While we include these allow lists in the repositories, you can generate them on your own by:

1. Opening the target's ELF file in IDA
2. While loading the binary, choosing ARMv7-M as the processor option
3. Running scripts/idapython/idapy_dump_valid_basic_block_list.py which is included in the fuzzware repository.
4. Execution function *dump_bbl_starts_txt()*.

You should now find a *valid_basic_blocks.txt* file next to the opened ELF file.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

[1] Note that we only include an automated experiment setup for Fuzzware, and not for rehosting frameworks we compare against.





A Artifact Appendix

A.1 Abstract

Our artifact includes three major parts: hardware-free device driver fuzzer, modified PANDA/QEMU with full-system concolic tracing support and concolic code exploration scripts. The code require a 64-bit x86 system with clean Ubuntu 20.04 install.

A.2 Artifact check-list (meta-information)

- **Compilation: It's best to use the default Ubuntu 20.04 compilers**
- **Run-time environment: Linux (preferably Ubuntu 20.04)**
- **Hardware: 64-bit x86 computer**
- **How much disk space required (approximately)?: 40G**
- **How much time is needed to prepare workflow (approximately)?: 30 min**
- **How much time is needed to complete experiments (approximately)?: several hours to days depending on number of drivers and duration of fuzzing session you want to run**
- **Publicly available (explicitly provide evolving version reference)?: https://github.com/messlabnyu/DrifuzzProject/tree/d0b9edfa364c2f9fe45d4b63c0ad9f62dca0bfc9**

A.3 Description

A.3.1 How to access

Clone source from https://github.com/messlabnyu/DrifuzzProject/. Or you can obtain our docker image from docker hub https://hub.docker.com/repository/docker/buszk/drifuzz-docker.

A.3.2 Hardware dependencies

64-bit x86 machine.

A.3.3 Software dependencies

Ubuntu 20.04 for building from source. Any Linux distro should be fine for running the Docker image.

A.4 Installation

```
git clone https://github.com/messlabnyu/↙
    DrifuzzProject.git
cd DrifuzzProject \&\& ./build.sh 2>\&1 |tee ↙
    build.log
```

A.5 Experiment workflow

In top-down perspective, our work invokes a golden seed search script to generate quality initial seeds. Then, we can run the fuzzing tool with the generated seed to increase coverage. Because our initial seed has solved many roadblocks, using it tends to find better coverage tank starting with random seed (e.g. Agamotto's approach). To find the golden seeds, we leverage concolic execution and forced execution to find and tackle roadblocks incrementally to increase coverage.

A.6 Evaluation and expected results

Evaluation should show that Drifuzz is able to perform concolic tracing in device driver execution and our golden seed search algorithm is able to provide a good initial seed resulting more code coverage.

A.6.1 Prerequisite

After installation, please check if the following files are created correctly. If any of the file was not created properly, please check the build log and script to triage the problem.

```
cd ~/DrifuzzRepo/Drifuzz
ls image/buster.img
ls panda-build/x86_64-softmmu/panda-system-↙
    x86_64
ls panda-build/x86_64-softmmu/panda/plugins/↙
    panda_taint2.so
ls linux-module-build/vmlinux
```

A.6.2 Concolic Tracing

Note: USB targets are supported with "–usb" flag in ./snapshot_helper.py and ./concolic.py.

```
cd ~/DrifuzzRepo/drifuzz-concolic

# Create a driver specific snapshot
./snapshot_helper.py ath9k
ls work/ath9k/ath9k.qcow2 # should exists

# Run concolic script with random input
head -c 4096 /dev/urandom >rand
./concolic.py ath9k rand
cat work/ath9k/drifuzz_path_constraints # path ↙
    constraints
cat work/ath9k/drifuzz_index # accessed MMIO/DMA
ls work/ath9k/out # generated inputs with ↙
    flipped branch

# Understand the concolic result
head work/ath9k/drifuzz_path_constraints
# Get the program counter of the first symbolic ↙
    branch
head work/ath9k/drifuzz_path_constraints |grep ↙
    PC | awk '{print␣$6}'
# Use addr2line script to get stack trace
./addr2line.py ath9k [program counter]
```

A.6.3 Golden seed

Note: you may encounter a bug that consumes all available disk space. In that case, run du tool to find and remove the files. If you

use provided docker image, deleting the container and retrying a new random seed might solve the problem.

Note: if you run into an AssertionError for "Cannot find a feasible path for given model" for the first ./concolic.py run, it seems that PANDA's concolic tracing is not work. Please double check that you have a snapshot in good standing and are able to run concolic tracing. If that fails, changing a seed is reported to work in the situation.

Note: USB targets are supported with "–usb" flag in ./search_greedy.py.

```
cd ~/DrifuzzRepo/drifuzz-concolic
# Run the golden seed script (takes a hour or so)
./search_greedy.py ath9k rand 2>&1|tee search_ath9k.log
ls work/ath9k/out/0 # generated seed
```

Fields below can be derived from generated log to compare with Table 2 from paper.

- *#blocking branches* = *#iterations* – 1
- *#symbolic branches* = *sizeof* (*last branches list*)

A.6.4 Fuzzing

```
cd ~/DrifuzzRepo/Drifuzz

# Fuzz ath9k with random seed on 4 cores
fuzzer/drifuzz.py -D -p 4 seed/seed-random work/ath9k ath9k
# Ctrl^C once to stop

# Reproduce a generated input
scripts/reproduce.sh ath9k work/ath9k/ work/ath9k/corpus/payload_1

# Process stacktrace when you see a crash
scripts/decode_stacktrace.sh crash.log
```

We also provide some helpful scripts to combine our golden seed and concolic support with our fuzzer. Note: You need to run the golden seed generation script before running some of the following scripts.

```
cd ~/DrifuzzRepo/Drifuzz

# Fuzzing random input without concolic support
scripts/run_random.sh ath9k
# Fuzzing golden seed with concolic support
scripts/run_conc_model.sh ath9k
```

A.6.5 Coverage comparison

Get the coverage metric from the fuzzing sessions. The result should show that the second session should have better coverage than the first. The detailed number will differ because of time of fuzzing period and non-determinism in fuzzing.

```
tail -n1 work/work-ath9k-random/evaluation/data.csv |awk -F';' '{print $16}'
tail -n1 work/work-ath9k-conc-model/evaluation/data.csv |awk -F';' '{print $16}'
```

A.7 Notes

Details of how to run each part of Drifuzz are shown in GitHub page.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

Our artifact is the code to create an ultra-wide band (UWB) high-Rate pulse repetition frequency (HRP) sniffer that generates accurate timestamps and forwards timestamps and UWB frames to Wireshark. We used this sniffer to identify timings in UWB ranging sequences and to attack the frames using the Ghostpeak attack. The artifact includes all necessary source code to run it on a recent DWM3000EVB from Decawave attached to a NUCLEO-F429ZI. A different board can be used, but the speed may suffer due to different SPI speeds.

We do not publish sample code for the attacks demonstrated in our paper, since this would violate German laws and might allow malicious actors to enter a system secured by UWB distance ranging.

A.2 Artifact check-list (meta-information)

- **Algorithm:**
- **Compilation: For the UWB sniffer we use the free STM32CubeIDE. The host a Python script.**
- **Binary: We do not include binaries, since the configuration needs to be changed depending on the UWB channels to listen on.**
- **Run-time environment: The STM32CubeIDE runs on Linux, macOS and Windows**
- **Hardware: We use a DWM3000EVB as the UWB receiver and attach it to a NUCLEO-F420ZI. The NUCLEO needs some slight hardware modifications according to the manufacturer Decawave.**
- **Execution: To actually sniff UWB signals some properties about the signals are needed: The channel, the preamble code and the start of frame delimiter (SFD) used.**
- **Output: Using the sensniff Python script the sniffer reports rx accurate timestamps and received frames**
- **Experiments: We do not apply for the reproducibility badge**
- **How much disk space required (approximately)?: 500KB**
- **How much time is needed to prepare workflow (approximately)?: 2 hours**
- **Publicly available: `https://github.com/seemoo-lab/uwb-sniffer`**
- **Code licenses: MIT License**
- **Archived: `https://github.com/seemoo-lab/uwb-sniffer/releases/tag/v1.0`**

A.3 Description

Our sniffer includes the necessary source code for the UWB board and the host machine to receive and process frames. Furthermore, we include a manual and a YouTube video on how to get started.

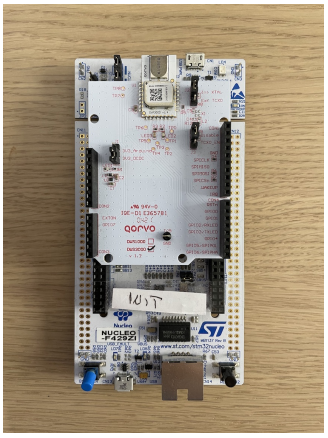


Figure 1: Shows a NUCELO-F429ZI with the DWM3000EVB attached.

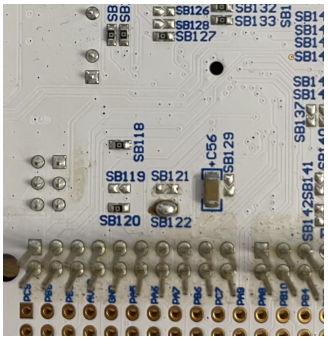


Figure 2: Necessary modifications on the NUCLEO board.

A.3.1 How to access

The artifact is available on GitHub: `https://github.com/seemoo-lab/uwb-sniffer/tree/usenix22-artifact-evaluation`.

Furthermore, we provide a YouTube video that show how to setup the sniffer and how to run it: `https://youtu.be/akCwyHqgbhY`.

A.3.2 Hardware dependencies

To run it we require the NUCELO-F429ZI and the DWM3000EVB attached to the NUCLEO as shown in Figure 1.

The NUCELO-F429ZI needs to be slightly modified to behave correctly when the DWM3000EVB is attached. These modifications are not necessary for nRF boards. Remove solder on SB121 and solder SB122. These modifications are shown in Figure 2.

A.3.3 Software dependencies

We use the STM32CubeIDE to compile and flash the attached NUCELO. We use Python to run a host script that receives input from the UWB Sniffer.

A.3.4 Data sets

N/A

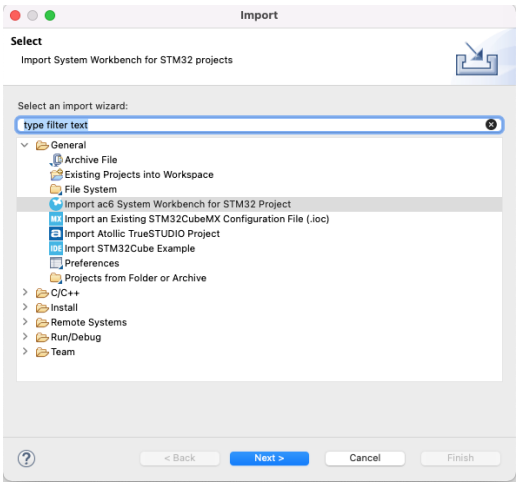


Figure 3: Importing a project in STM32CubeIDE.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

None.

A.4 Installation

Download the STM32CubeIDE and install it: https://www.st.com/en/development-tools/stm32cubeide.html

Download the software samples from Decawave, which include the API to communicate with the DWM3000EVB. Due to license issues, we cannot host it on our GitHub. https://www.decawave.com/wp-content/uploads/2022/03/DW3xxx_XR6.0C_24Feb2022.zip

1. Open a new workspace in the STM32CubeIDE
2. Go to File → Import → General → "Import ac6 System Workbench for STM32 Project" (see Figure 3)
3. Select the root folder of the sample project and import the project
4. Accept to convert the project to the new format
5. Now you can build and run the examples
6. Make sure that the examples build without an error

A.4.1 Integrate the sniffer

1. Copy all source files from this project to the root folder of the DWM3000 sample code
2. Overwrite the main.c with the one in this project
3. Compile the project
4. Run it on an NUCLEO-F429ZI

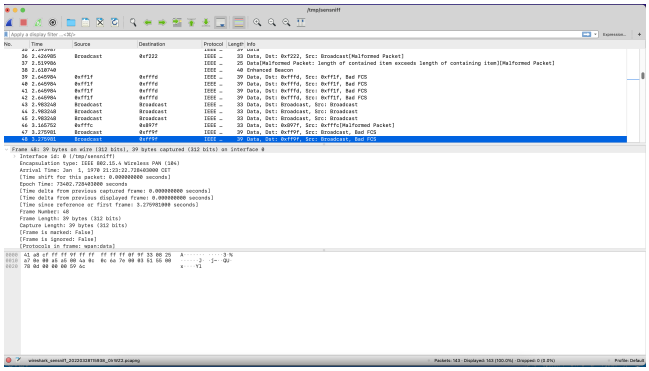


Figure 4: Screenshot of Wireshark with received frames.

A.4.2 Configure the sniffer

UWB has a variety of available configurations: channel, preamble code, data rate, sts mode, and sts length. Most of them have to be known in advance to sniff a communication. In most cases these values can be identified through means of reverse-engineering. For iOS UWB communication, we use the iOS system logs from the nearbyd to identify those values. The values can be changed in the `uwb_sniffer.c` file in the `config` struct.

In the current implementation the sniffer also transfers frames with incorrect headers or frame lengths to the host. So make sure to check the Wireshark output if it is correct. An incorrect configuration leads to long and incorrect frames where the STS or pars of the preamble will be interpreted as data.

A.5 Experiment workflow

With the sniffer any UWB communication following the IEEE802.15.4-2020 HRP standard can be sniffed. This includes frames and accurate timestamps. For this the sniffer needs to be configured as described in Section A.4.2.

With the STM32CubeIDE you can flash the NUCLEO with the attached DWM3000EVB board. When powered on it will immediately start sniffing and trying to forward the packets to the attached computer. To receive the frames on the computer it's necessary to run the provided python script.

```
$ python3 sensniff.py -a
```

The script will try to automatically detect the connected device. If this fails (due to different OS support), we recommend passing the right port directly:

```
$ python3 sensniff.py -d /dev/cu.usbmodem230d
```

Then launch Wireshark on the same computer and add a new pipe at `/temp/sensniff`. When listening to this pipe in Wireshark all frames received by the UWB sniffer will appear here. This includes accurate timestamps when the frame has been received. Figure 4 is a screenshot of a running Wireshark instance that receives UWB frames.

A.6 Evaluation and expected results

In our paper we state that we are able to run the attack Ghostpeak that achieves distance reductions in UWB ranging environments using the modern IEEE 802.15.4z standard. The attack works against the

Apple U1 chip in combination with any other compatible UWB chip (DW3000 from Qorvo/Decawave and SR150 from NXP). We do not publish the attack code, but provide our code for the UWB sniffer, which we used to measure accurate timestamps of the attack and analyze ranging sequences. The sniffer also forwards UWB frames to the host machine which can then display them in Wireshark.

Most U1 to U1 ranging scenarios can be monitored using the sniffer. Depending on the devices used and the ranging sequence in use the sniffer may need to be modified.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

Secure inference allows a model owner (or, the server) and the input owner (or, the client) to perform inference on machine learning model without revealing their private information to each other. Recently, Lehmkuhl *et al.* proposed a secure inference system, MUSE, in client malicious threat model. In our paper titled "SIMC: ML Inference Secure Against Malicious Clients at Semi-Honest Cost", we design and build SIMC, a new cryptographic system for secure inference in client malicious threat model.

In this artifact, we implement our proposed system SIMC. Using this implementation, we show that SIMC has $23-29\times$ lesser communication and is up to $11.4\times$ faster than MUSE, for benchmarks considered by MUSE. SIMC obtains these improvements using a novel protocol for non-linear activation functions (such as ReLU) that has $> 28\times$ lesser communication and is up to $43\times$ more performant than MUSE.

In this article, we summarize the system requirement, installation and building process, and finally, the execution process in order to obtain the performance numbers reported in our paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** SIMC (Secure Inference Against Malicious Client) protocol.
- **Program:** Implementation in C++ (https://aka.ms/simc).
- **How much disk space required (approximately)?:** 16GB.
- **How much time is needed to prepare workflow (approximately)?:** 3 hours.
- **How much time is needed to complete experiments (approximately)?:** 20 minutes
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.

A.3 Description

A.3.1 How to access

Access the github repo using link: https://aka.ms/simc (commit id: 2a5fd092b52427cc9cac55b36ec50ae43ecee6be).

A.3.2 Software dependencies

Install Eigen3, SEAL and emp-toolkit repositories. See Installation steps for more details.

A.4 Installation and Compilation

1. Create parent directory `msi-code`

   ```
   mkdir msi-code && cd msi-code
   ```

2. To install Eigen3 do

   ```
   sudo apt-get update -y
   sudo apt-get install -y libeigen3-dev
   ```

3. Follow the installation steps of [emp-toolkit/emp-sh2pc].
4. Clone this repo in the parent directory `msi-code`.
5. Install SEAL 3.64
   (a) Clone SEAL 3.6 repo in the parent directory `msi-code`.
   (b) Execute

   ```
   cd SEAL
   git checkout 3.6.4
   mkdir build && cd build
   cmake ..
   make -j
   sudo make install
   ```

6. In `msi-code`, go to `emp-tool` and do `git checkout df363bf30b56c48a12c352845efa3a4d8f75b388`.
7. Next, go to `emp-ot` in `msi-code` and do `git checkout 3b21d6314cb1e7d8dbb9bb1f1ed80261738e4f4c`.
8. For multi-threading support, go to `emp-tool` and run the following

   ```
   cmake . -DTHREADING=ON
   make -j
   sudo make install
   ```

9. Do the same for `emp-ot` repository.
10. Finally, do the same in our (`simc`) repository.

A.5 Experiment workflow

The protocol is run between two parties. Open two terminal windows and run the following test files from path `msi-code/simc`:

A.5.1 Run Neuralnet Benchmarks

1. Fully-connected Layer: In one terminal run `bin/test_msi_linearlayer 1 0.0.0.0 <port_no> 44 <neural_network>` and in other terminal run `bin/test_msi_linearlayer 2 <server_ip_address> <port_no> 44 <neural_network>`.
2. Convolution Layer: In one terminal run `bin/test_msi_convlayer 1 0.0.0.0 <port_no> 44 <neural_network>` and in other terminal run `bin/test_msi_convlayer 2 <server_ip_address> <port_no> 44 <neural_network>`.
3. Non-Linear Layer (ReLU): In one terminal run `bin/test_msi_relu_final 1 0.0.0.0 <port_no> 44 <neural_network>` 0 0 <num_threads> and in other terminal run `bin/test_msi_relu_final 2 <server_ip_address> <port_no> 44 <neural_network>` 0 0 <num_threads>.
4. Average Pool Layer: In one terminal run `bin/test_msi_average 1 0.0.0.0 <port_no> 44 <neural_network>` and in other terminal run `bin/test_msi_average 2 <server_ip_address> <port_no> 44 <neural_network>`.

Here, the first parameters 1 and 2 denote the ID of the participating party. `<server_ip_address>` denotes the ip address of the server machine and set `<neural_network>`=1 for MNIST and `<neural_network>`=2 for CIFAR-10. See Figure 1 for examples.

A.5.2 Run Neuralnet Micro-benchmarks

See Figure 2 for instructions and examples to run micro-benchmarks. Note that, for different system-configuration, different number of threads may provide best performance for given number of ReLUs.

A.6 Evaluation and expected results

To obtain performance numbers of our protocol that were used to generate the plot of Figure 7 of our paper, follow instructions in Section A.5.2.

Follow instructions in Section A.5.1, and then aggregate the observed runtime and communication cost across all the layers to obtain performance numbers of Tables 1, 2 and 3 of our paper. If the protocols are run in similar system setting as ours, the observed runtime will be similar to what has been reported in paper.

```
Fully connected Layer:
Terminal 1: bin/test_msi_linearlayer 1 0.0.0.0 31000 44 1
Terminal 2: bin/test_msi_linearlayer 2 <server_ip_address> 31000 44 1

Convolution Layer:
Terminal 1: bin/test_msi_convlayer 1 0.0.0.0 31000 44 1
Terminal 2: bin/test_msi_convlayer 2 <server_ip_address> 31000 44 1

Non-Linear Layer (ReLU):
Terminal 1: bin/test_msi_relu_final 1 0.0.0.0 31000 44 1 0 0 8
Terminal 2: bin/test_msi_relu_final 2 <server_ip_address> 31000 44 1 0 0 8

Average Pool Layer:
Terminal 1: bin/test_msi_average 1 0.0.0.0 31000 44 1
Terminal 2: bin/test_msi_average 2 <server_ip_address> 31000 44 1
```

Figure 1: Run Neuralnet Benchmarks Examples

```
Terminal 1: bin/test_msi_microbenchmark 1 0.0.0.0 31000 44 <benchmark_choice>
            <num_relus> <#threads>
Terminal 2: bin/test_msi_microbenchmark 2 <server_ip_address> 31000 44 <benchmark_choice>
            <num_relus> <#threads>


Input Parameters:
1. <server_ip_address>: IP Address of Server.
2. <benchmark_choice>: 0 - ReLU6, 1 - ReLU.
3. <num_relus>: Number of ReLUs
4. <#threads>: Number of threads


if <num_relus> <=2, set <#threads>=1,
else if <num_relus> <=4, set <#threads>=2,
else if <num_relus> <=16, set <#threads>=4,
else if <num_relus> >16, set <#threads>=8.

Example:
Terminal 1: bin/test_msi_microbenchmark 1 0.0.0.0 31000 44 0 16384 8
Terminal 2: bin/test_msi_microbenchmark 2 <server_ip_address> 31000 44 0 16384 8
```

Figure 2: Run Neuralnet Micro-benchmarks



A Artifact Appendix

A.1 Abstract

The artifact is based on PyTorch and requires GPU support. We implement our MIA defense SELENA, which consists of two components: Split-AI (Algorithm 1) and Self-Distillation. We also provide the implementation of prior defenses Adversarial Regularization [30] and MemGuard [21]. The artifact can reproduce experimental results in the main body, i.e., Table 2.

Our source code is available at https://github.com/inspire-group/MIAdefenseSELENA/tree/39428e763566a8276d82e1c0fe91bbaaddb84bfb.
We further provide a detailed guide for evaluating our artifact at https://github.com/inspire-group/MIAdefenseSELENA/blob/39428e763566a8276d82e1c0fe91bbaaddb84bfb/misc/reproducibility.md.

A.2 Artifact check-list (meta-information)

- **Algorithm:** We implement our defense SELENA, which consists of two components: Split-AI (Algorithm 1) and Self-Distillation from Split-AI. We also provide comparison with prior defenses: undefended model, adversarial regularization [30] and MemGuard [21].
- **Program:** N/A.
- **Compilation:** N/A.
- **Transformations:** N/A.
- **Binary:** N/A.
- **Model:** 4-layer fully connected neural network and ResNet-18.
- **Data set:** Purchase100, Texas100, CIFAR100. They are publicly available benchmark datasets.
- **Run-time environment:** We test our artifact using anaconda virtual environment on Linux.
- **Hardware:** Requires one GPU.
- **Run-time state:** N/A.
- **Execution:** N/A.
- **Security, privacy, and ethical concerns:** N/A.
- **Metrics:** Membership inference attack accuracy. This is defined in Section 3.1. Random guess baseline is 50%.
- **Output:** We output results (classification accuracy and MIA accuracy) and intermediate results to console.
- **Experiments:** We provide reproducing instructions including commands.
- **How much disk space required (approximately)?:** Datasets take around 4 GB. Each model weight takes less than 100 MB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** The time range of defense for each experiment varies from a few minutes to around 30 hours (from scratch). See Table 3 and Table 4 for a reference. The time range of MIA attacks is approximately a few minutes for direct single-query attacks and data augmentation attacks, 4 $\sim$ 8 h for flip noise attack, 20 h for boundary distance attacks. The time of adaptive attacks is similar to the time of run SELENA defense.
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/inspire-group/MIAdefenseSELENA/tree/39428e763566a8276d82e1c0fe91bbaaddb84bfb.
- **Code licenses (if publicly available)?:** MIT License.
- **Data licenses (if publicly available)?:** N/A.
- **Workflow frameworks used?:** N/A.
- **Archived (explicitly provide DOI or stable reference)?:** N/A.

A.3 Description

A.3.1 How to access

We host our source code on GitHub at https://github.com/inspire-group/MIAdefenseSELENA/tree/39428e763566a8276d82e1c0fe91bbaaddb84bfb.
We further provide a detailed guide for evaluating our artifact at https://github.com/inspire-group/MIAdefenseSELENA/blob/39428e763566a8276d82e1c0fe91bbaaddb84bfb/misc/reproducibility.md.

A.3.2 Hardware dependencies

The artifact requires 1 CPU and 1 GPU.

A.3.3 Software dependencies

The artifact is based on Python, PyTorch, TensorFlow and other Python packages. All packages can be easily installed with pip; we provide a list of required packages in requirement.txt.

A.3.4 Data sets

We use three publicly available datasets in our evaluation: Purchase100, Texas100, CIFAR100. See our reproducing instructions for more details.

A.3.5 Models

The 4-layer fully connected neural network is for Purchase100/Texas100, which is widely used in prior MIA defenses [21, 30]. The ResNet-18 model for CIFAR100, which is widely used in image classification tasks.

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Steps 1-3 can also be done in the Anaconda environment.

1. Install Python [help link] (or Anaconda ([help link]): conda create -n myenv python=3.8.5).
2. Install GPU-compatible PyTorch [help link] and TensorFlow. [help link] (or Anaconda: GPU-compatiable PyTorch [help link] and TensorFlow [help link]).
3. Install other Python dependencies [help link].
4. Clone the source code from `https://github.com/inspire-group/MIAdefenseSELENA/tree/39428e763566a8276d82e1c0fe91bbaaddb84bfb`.
5. Follow the preparation steps in Getting Started [help link].

A.5 Experiment workflow

Our defense is implemented in $datasetname/SELENA folder. After preparing the initial dataset and environments, we first need to run $datasetname/data_partition.py to generate the npy files for member/nonmember sets to train/eval MIA attacks. We also need to generate the non-model indices for training set via $datasetname/SELENA/generation10.py. Then we need to train Split-AI model by $datasetname/SELENA/Split-AI/train.py. Next, we need to train the Self-Distillation model by $datasetname/SELENA/Distillation/train.py. To evaluate the protected model from Self-Distillation by membership inference attacks, we need to run $datasetname/SELENA/Distillation/eval.py (eval_cw.pyleval_aug.py). See reproducing instructions for more details. We can read the training/test accuracy for the classification model, and the membership inference attack accuracy from the console, which is the corresponding result of Table 2 in the paper.

A.6 Evaluation and expected results

Our main claim is that our defense SELENA achieves a better trade-off between empirical membership privacy and utility compared to the state of the art MIA defenses [21,30]. This claim is supported by Table 2 of our paper. We can use commands listed in our reproducing instructions to generate our key results including the classification accuracy and MIA attack accuracy of our defense as well as prior MIA defenses [21, 30]. For accuracy on training set, see the corresponding classification accuracy for 'train'. For accuracy on test set, see the corresponding classification accuracy for 'test'. For direct single-query attacks, see 'Best direct single-query attack acc: '. For label-only attacks, see 'Best label-only attack at flip:' or 'CW attack:' or 'Augmentation attack:'. For adaptive attacks: see 'BEST ATTACK ACC:'.

The reported number should be consistent with Table 2 in the main body. It's possible to have around $0\% \sim 2\%$ mismatches due to some randomness.

A.7 Experiment customization

Our source code provides an easy way to customize the experiment. The main algorithm in our SELENA defense is to generate non_model indices and perform adaptive inference on Split-AI, which can be easy adapted to datasets/models not listed in the source code. The parameters K and L can also be changed via flag `-K` and `-L` when needed.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

The artifact is an open-source Python library that implements a novel framework to evaluate the privacy-utility trade-off of synthetic data publishing and to compare it to that of traditional sanitisation techniques. The library provides implementations of two privacy attacks to evaluate privacy gain with respect to the risk of linkability and inference. It further includes implementations of five example generative models, three standard models and two models trained under formal privacy guarantees.

The artifact contains experiment scripts and configuration files to reproduce some of the results presented in the paper. In particular, the example runs described in the README allow the user to partially reproduce the graphs of Section 6 that compares the privacy-utility trade-off of (differentially private) synthetic data to traditional anonymisation.

A.2 Artifact checklist

- **Algorithm:** The privacy (and utility) games introduced in the paper are implemented in the corresponding command line interface (`cli` files.)
- **Data set:** The repository contains a copy of the cleaned-up and pre-processed dataset used for the main set of experiments. The dataset and the required metadata can be found in the data folder under `texas.csv` and `texas.json`, respectively.
- **Run-time environment:** Synthetic Data is also distributed as a ready-to-use Docker image containing Python 3.9 and CUDA 11.4.2, along with all dependencies required by Synthetic Data.
- **Hardware:** When running evaluations for either the `PATE-GAN` or `CTGAN` model it is useful to have a GPU at hand. This significantly speeds up the execution. However, they are not needed for running the example experiments.
- **Execution:** The README includes instructions about how to run three example experiments. The evaluation under the linkage risk model is the most compute- and memory-intensive. On a machine with an Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz with 2 cores (with hyperthreading) this should take around 1h15m.
- **Output:** The example experiments produce output files in a `json` format and can be parsed with the functions provided in `utils/analyse_rersults`. We include a simple jupyter notebook that allows to visualise and analyse the results.
- **Experiments:** The repository includes experiment configuration for three key experiments. See further below of the README of the repository.
- **How much disk space required (approximately)?:** If using the dockerised deployment, its image requires ~14GB of disk space. The experiment outputs need
- **How much time is needed to prepare workflow (approximately)?:** <1h (but strongly depends on the bandwidth of the connection used to pull the Docker image).
- **How much time is needed to complete experiments (approximately)?:** This depends on the compute power available. On a machine with an Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz with 2 cores (with hyperthreading) it should take 3h to run all example experiments.
- **Publicly available?:** The code is publicly available at https://github.com/spring-epfl/synthetic_data_release/tree/v1.1
- **Code licenses (if publicly available)?:** The code is distributed under a BSD-3-Clause License.
- **Data licenses (if publicly available)?:** See https://www.dshs.texas.gov/THCIC/Hospitals/Download.shtm

A.3 Description

The library has two main classes: `GenerativeModels` and `PrivacyAttacks`. For both classes we define a parent class that determines the core functionality that objects of the class need to implement.

`GenerativeModel` provides two main functions. `GM.fit(R)` is called with a raw dataset `R` as input and implements the model's training procedure. `GM.sample(m)` generates a synthetic dataset `S` of size `m`. The library enables easy integration of existing model training procedures. `GM.fit` simply wraps any existing training algorithm and exposes the appropriate API endpoints.

`PrivacyAttack` objects have two functions: `PA.train` and `PA.attack`. `PA.train` trains the adversary's guess function and needs to be run before calling `PA.attack`. `PA.attack(S)`, takes a dataset `S` and outputs a guess about a secret value. In our implementation, we instantiate `PrivacyAttack` with two attacks, a linkage adversary and an attribute inference attack.

The library also includes procedures to estimate the privacy gain of synthetic and sanitised data publishing. These procedures can be found in the corresponding `cli` files.

A.3.1 How to access

The code, some example data and experiment configuration files are publicly available at https://github.com/spring-epfl/synthetic_data_release/tree/v1.1.

A.3.2 Dependencies

For your convenience, Synthetic Data is also distributed as a ready-to-use Docker image containing Python 3.9 and CUDA 11.4.2, along with all dependencies required by Synthetic Data.

Note: This distribution includes CUDA binaries, before downloading the image, ensure to read its EULA and to agree to its terms.

A.3.3 Data sets

The repository contains a copy of the cleaned-up and pre-processed dataset used for the main set of experiments, the Texas hospital dataset. The dataset and the required metadata can be found in the data folder under `texas.csv` and `texas.json`, respectively.

The Texas Hospital Discharge dataset is a large public use data file provided by the Texas Department of State Health Services. The dataset we include here for the experiments consists of 100,000

records uniformly sampled from a pre-processed data file that contains patient records from the year 2013 and 2014. We retain 18 data attributes of which 11 are categorical and 7 continuous.

A.4 Installation

Synthetic Data can either be installed from scratch or run in a dockerised environment. If you want to use the ready-to-use docker container, pull the image and run a container (and bind a volume where you want to save the data):

```
docker pull springepfl/synthetic-data:latest
docker run -it -rm -v "$(pwd)/output:/output"
springepfl/synthetic-data
```

The Synthetic Data directory is placed at the root directory of the container.

```
cd /synthetic_data_release
```

You should now be able to run the examples without encountering any problems.

A.5 Experiment workflow

We provide three example experiments and their configurations to reproduce some of the key claims presented in Section 6 of the paper.

Privacy gain with respect to linkability. First, to run a privacy evaluation with respect to the privacy concern of linkability you can run

```
python3 linkage_cli.py -D data/texas -RC
tests/linkage/runconfig.json -O tests/linkage
```

The results file produced after successfully running the script will be written to `tests/linkage` and can be parsed with the function `load_results_linkage` provided in `utils/analyse_results.py`. A jupyter notebook to visualise and analyse the results is included at `notebooks/Analyse Results.ipynb`.

Privacy gain with respect to inference. To run a privacy evaluation with respect to the privacy concern of inference you can run

```
python3 inference_cli.py -D data/texas -RC
tests/inference/runconfig.json -O tests/inference
```

The results file produced after successfully running the script can be parsed with the function `load_results_inference` provided in `utils/analyse_results.py`. A jupyter notebook to visualise and analyse the results is included at `notebooks/Analyse Results.ipynb`.

Average machine learning utility. To run a utility evaluation with respect to a simple classification task as utility function run

```
python3 utility_cli.py -D data/texas -RC
tests/utility/runconfig.json -O tests/utility
```

The results file produced after successfully running the script can be parsed with the function `load_results_utility` provided in `utils/analyse_results.py`. The jupyter notebook contains code for visualising the results.

A.6 Evaluation and expected results

Privacy gain with respect to linkability. This experiment allows you to compare the privacy gain for five outlier records from the Texas dataset with respect to the risk of linkability for three different privacy mechanisms: traditional sanitisation (`SanitiserNHSk10`), synthetic data produced by a standard Bayesian Network (`BayNet`) and a differentially private version of this model (`PrivBay`). The differentially private model is trained with a privacy parameter of $\varepsilon = 1.0$. All other model hyperparameters match the ones used in the paper.

After loading the results into the notebook named `Analyse Results.ipynb`, you can inspect the per-target privacy gain for five outlier records from the Texas dataset under a linkage attack with varying feature sets. You should observe that, as described in the paper, the privacy gain of most targets is larger under the `BayesianNet` model compared to traditional sanitisation and further increases if the synthetic data is sampled from the differentially private model (compare with Fig.4 in Section 6.1 in the final version of the paper). You can choose under which attack feature set you want to compare the targets' privacy gain.

Note: Due to the sampling uncertainty and randomness of the attack and generative model training process, you should expect slight variations between the observed privacy gain and the exact values reported in the final publication. Furthermore, the observed variance of the per-record privacy gain is likely larger than the one reported in the final publication. This is because the privacy game is run for a smaller number of iterations to reduce the computation time of the experiments. To reduce the reported standard variation, you can modify the parameter `nIter` in the configuration file `tests/linkage/runconfig.json`.

Privacy gain with respect to inference. Similarly, the privacy gain of the same target records under the same models with respect to the risk of inference can be evaluated with the results file written to `tests/inference/`. The results should be comparable to the data presented in Fig. 6 of Section 6.2.

Utility loss under a classification task. This experiment allows you to compare the utility loss of sanitised and synthetic data publishing under a simple classification task as utility function. The details of this experiment are described in Section 6.3.1. Here, we want to compare the accuracy of a machine learning classifier trained on a sanitised or synthetic dataset to that of a classifier trained on the raw data. You should observe how training on data with a higher privacy gain, i.e., (differentially private) synthetic data, leads to a decline in the model's accuracy compare to the raw data.

A.7 Experiment customization

To change the evaluation parameters, you can modify the `runconfig.json` files in the corresponding experiment folders. For instance, to change the size of the raw and synthetic datasets, respectively, you can modify the parameters `sizeRawT` and `sizeSynT` for each experiment file.

To evaluate the privacy gain of synthetic data publishing under a different generative model, a new `GenerativeModel` class has to be integrated. See A.3 for more details on the implementation of this class. If you want to run the evaluation on an entirely new dataset, a metadata file in `.json` format is necessary. You can use the `texas.json` metadata file as a template.



A Artifact Appendix

A.1 Abstract

We demonstrate how targeted deanonymization attacks performed via the CPU cache side channel can circumvent browser-based defenses. The attack framework we show is able to overcome the limitations of prior work, such as assumptions on the existence of cross-site leaks. As a result of this attack, the attacker is able to learn whether a specific individual visits the attacker-controlled website – a potentially serious privacy violation.

When a user visits the attacker-controlled website, the website uses an iframe, popunder, or tabunder to request a resource from a third-party website (i.e., the "leaky resource"). The response to this request, as well as the cache activity it generates in the user's system, differs depending on the user state on the third-party website. An attacker monitoring the CPU cache side channel can analyze the cache patterns and learn whether the leaky resource was loaded successfully in the browser or not, and use this information to learn the identity of the visiting user. The attack can be scaled to identify thousands of users.

The artifact repository is hosted at GitHub and evaluations are performed on Google Colab. The reviewers should run the provided scripts on Google Colab. To support the feasibility of the attacks and the defense proposed in the paper, the results should be similar to Figure 5 and Table 1, 2 and 6 of the paper.

A.2 Artifact check-list (meta-information)

- **Data set: dataset.zip**
- **Run-time environment: Google Colab**
- **How much disk space required (approximately)?: 200MB**
- **How much time is needed to complete experiments (approximately)?: one hour**
- **Archived (explicitly provide DOI or stable reference)?: https://github.com/leakuidatorplusteam/artifacts.git commit ID: 78bae165e0dbcdeb245b19a1f5b75a191de92fc3**

A.3 Description

We submit for **Artifacts Available**, **Artifacts Functional** and **Results Reproduced** badges.

A.3.1 How to access

```
git clone git@github.com:
leakuidatorplusteam/artifacts.git

cd artifacts
git checkout 78bae165e0dbcdeb245b19a1f5b75a191de92fc3
```

A.3.2 Hardware dependencies

To collect additional traces, one of the following systems are required:

```
Dell Latitude - Intel Core i7 7820HQ

Mac mini - Apple M1 8-Core

MacBook Pro - Intel Core i7 3540M
```

A.3.3 Software dependencies

To collect additional traces, one of the following systems are required:

```
Windows 10 Pro 20H2 - Chrome 96.0

macOS Big Sur 11.4 - Chrome 96.0

macOS Catalina 10.15.7 - Safari 15.0
```

A.3.4 Data sets

dataset.zip file is available at the root directory of the artifacts repository hosted at GitHub.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Git software can be used to get local access to the repository. Reviewers need to have Google accounts to access Google Colab and also to share resources privately.

A.5 Evaluation and expected results

Here we describe the step by step instructions of two phases of the evaluation. In the first phase, we demonstrate how the dataset we already collected can be used to train the classifiers and determine the accuracy of attacks: [1] [2]

1. Open https://colab.research.google.com/

2. From your local copy of artifacts repository, upload the USENIX_Artifact_Evaluation/cache_demo.ipynb file in Google Colab and open it
3. On the left side of Google Colab interface, click on “Files”, then “Upload to session storage”, and choose the *dataset.zip* file from your local copy of the artifacts repository
4. From the menu on top of Google Colab interface, click on “Runtime”, then “Run all”, and wait until it is finished

After finishing these 4 steps, the reported results are as follows:

- The code block with comment starting with "## [Single Target Attacks]" shows the prediction accuracy on the dataset using LR (logistic regression classifier), MSE (mean squared error), and FastDTW (fast dynamic time warping). These results correspond to the results reported on Table 1 and 6
- The code block with the comment starting with "## [Chrome Android]" shows the results for experiments with Android Chrome
- The code block with the comment starting with "## [Old Defense (Leakuidator)]" shows the results for experiments with old defense prior to the modifications suggested in this paper
- The code block with the comment starting with "### [Multi Target Attacks]" shows the results for experiments with multi target attacks, reported in Table 2 of the paper
- The code block with the comment starting with "## [Average and Attack Accuracy plots]" correspond to Figure 5 of the paper.

In the second phase, we provide a step by step instruction to demonstrate how the attack page collects the cache traces and uses them for prediction. To run the attack from scratch, reviewers can collect traces using one of three systems Win-Chrome, Mac-Intel-Safari, or Mac-M1-Chrome detailed in Table 4 of the paper, using the respective attack pages at USENIX_Artifact_Evaluation directory. To customize the targeted deanonymization attack demo for a target user of your choice, do the following:

1. Login to the attacker Youtube account (e.g., attacker@gmail.com) at youtube.com
2. Upload two private videos of at least 1 second duration in the attacker Youtube account
3. Write down the identifier of the private videos you created, called **[video_id_1]** and **[video_id_2]**
4. Share **[video_id_1]** privately only with the targeted victim you’d like to track (e.g. victim@gmail.com) and **[video_id_2]** privately only with another attacker account (e.g. attacker_second_id@gmail.com)
5. Prepare the state dependent URL as follows: "https://www.youtube.com/embed/**[video_id_1]** ?rel=0&autoplay=1&mute=1"
6. Prepare the URL for the non-target state as follows: "https://www.youtube.com/embed/**[video_id_2]** ?rel=0&autoplay=1&mute=1"
7. Prepare two attack pages page_1.html and page_2.html and change the "State-Dependent-URL" string in the source code of the attack pages to these two URLs: page_1 points to the URL at step 5 and page_2 points to the URL at step 6
8. Host the attack pages on a web server (either local or remote). In particular, do not run the attack pages as local files (i.e., not served by a web server)
9. Log out of the attacker’s youtube account, and login to the victim’s youtube account
10. Open two tabs in the browser. The first tab points to page_1 resembling the target state, and the second tab points to page_2 resembling the non-target state. Record the traces first in the target tab, then in the non-target tab, then again target tab, then non-target tab, ..., and repeat this at least 100 times. (to make the experiment easier, instead of manually performing this experiment, customize and use the scripts at the "automation_scripts" folder)
11. Put the collected traces into the template.json file (100 target traces and 100 non-target traces)
12. Open https://colab.research.google.com/
13. Upload the USENIX_Artifact_Evaluation/test.ipynb file to Google Colab and open it
14. On the left side of Google Colab interface, click on “Files”, then “Upload to session storage”, and choose the template.json file that contains your collected traces
15. Set the sweep and interval parameters as suggested in the comments
16. From the menu on top of Google Colab interface, click on “Runtime”, then “Run all”, and wait until it is finished

After finishing these steps, an average plot is generated. It should be somewhat similar to Figure 5 in the paper, demonstrating the differences between the two states. Also, accuracy of the logistic regression classifier is reported.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



C Artifact Appendix

C.1 Abstract

Throughout the paper, the results obtained in summary statistics, statistical tests, and qualitative coding directly inform our observations on user behavior. To establish the validity of these findings, we provide the data and subsequent analysis to replicate all results reported in this paper. Specifically, we provide our collected data (for review only), our quantitative analysis via an R notebook, and the results of our qualitative analysis via an excel spreadsheet (for review only). Using these, one can recreate all results in tables, figures, statistical tests, and reported code counts throughout the paper.

C.2 Artifact Check-List (Meta-Information)

- **Data set:** Our collected user study data; non-public.
- **Run-time environment:** Tested on macOS 12.0 Monterey and Ubuntu 20.04.
- **Security, privacy, and ethical concerns:** Maintaining the confidentiality of participant data.
- **Metrics:** Perceived trustworthiness of a social media profile; log likelihood of accepting a social connection.
- **Output:** The artifact produces all result-containing tables, figures, and the code counts of the reported user answers.
- **Experiments:** Statistical and qualitative analysis of user responses.
- **How much disk space required (approximately)?:** 5 GBs.
- **How much time is needed to prepare workflow (approximately)?:** 60 minutes.
- **How much time is needed to complete experiments (approximately)?:** 2 minutes.
- **Publicly available (explicitly provide evolving version reference)?:** All scripts and code are made publicly available[16].
- **Code licenses (if publicly available)?:** University of Illinois/NCSA Open Source License.
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/JaronMink/DeepPhish/releases/tag/USENIX-22-artifact-evaluation

C.3 Description

C.3.1 How to Access

Along with various supplemental material, we make all the scripts and code used to analyze data and perform statistical tests publicly available[16].

As shown in Figure 10, the artifact is comprised of three main parts: (1) the 'data" folder which contains anonymized participant responses; (2) the R notebook "quantitative_analysis.Rmd" which provides the results reported in tables and figures throughout the paper with the expected output "expected_quantitative_analysis.pdf"; and (3) the spreadsheet "qualitative_codings.xlsx" with the script "codebook_kappa.py" which respectively contains the coded responses and calculates the inter-rater agreement of the codes. Additionally, the README.md provides a detailed overview of all files.

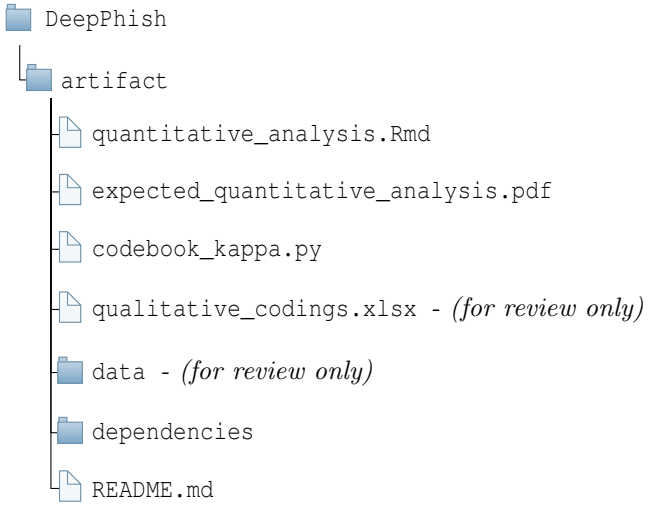


Figure 10: **Artifact File Structure** – We present the file structure of the artifact folder in the paper's supplemental materials[16]. The "qualitative_codings.xlsx" file and "data" folder contain sensitive participant data and thus are only provided for review.

C.3.2 Hardware Dependencies

This analysis requires approximately 5 GBs of disk space.

C.3.3 Software Dependencies

To run the quantitative analysis, we make use of R (4.1.2) RStudio (2021.09.2 Build 382), Pandoc (2.5) and a host of R libraries. We provide download scripts for the specific libraries and dependencies used in macOS and Ubuntu.

To perform the qualitative analysis, we use Python (3.8.10) and Pip (20.0.2) to run the Cohen's-Kappa calculation and Microsoft Excel (16.55) to view the coding spreadsheet (any .xlsx viewer will suffice).

C.3.4 Data Sets

We use the data collected in our user studies. While the data is provided to reviewers, to maintain participant privacy, we do not release this data publicly.

C.3.5 Models

We train our linear mixed-effects model (Section 4.1) and our logistic mixed-effects model (Section 4.2) on the

[16] https://github.com/JaronMink/DeepPhish

gathered user-study data via the R notebook "quantitative_analysis.Rmd".

C.3.6 Security, Privacy, and Ethical Concerns

While there is no inherent risk in our analysis, all participant-provided data should be treated with care. We took steps to anonymize all direct identifiers; however, due to the nature of user and qualitative responses, we cannot exclude the possibility of such data being used to deanonymize participants.

C.4 Installation

C.4.1 Quantitative Analysis

Installation time: ~45 minutes

The quantitative evaluation is performed in an R-notebook and thus requires various software libraries, frameworks, and system dependencies to support it.

Software. R, RStudio, and Pandoc are all publicly available and their instructions for version-specific installation can be found at their respective websites.

System Dependencies. As many R libraries require various system dependencies, we provide scripts to download the required dependencies for Ubuntu ("install_ubuntu_dependencies.sh") and macOS ("install_macos_dependencies.sh").

R Libraries. To install the R libraries used in the analysis, we provide an OS-independent bash script: "install_r_libraries.sh".

C.4.2 Qualitative Analysis

Installation time: ~5 minutes Python

Software. Python (3.8.10) and Pip (20.0.2) are both publicly available and their instructions for version-specific installation can be found at their respective websites.

Python Libraries. To install the utilized Python libraries, we provide a pip3-compatible requirements file: "requirements.txt".

C.5 Evaluation and Expected Results

C.5.1 Quantitative Analysis

Execution Time: ~2 minutes

The results from Section 3, Section 4, Section 5, and Appendix B are produced in the R notebook via the following steps:

1. Open the R Studio application or go to the assigned localhost port with a web-browser (default is 8787).
2. Open the notebook: "quantitative_analysis.Rmd"
3. Produce the results by selecting "Knit → Knit to PDF".
4. Once completed, you may view the produced PDF: "quantitative_analysis.pdf"

The PDF will contain the results for the tables, figures, and information found in the paper which directly inform Observations 1-4.

Section 3.6: Demographic background and time distributions of participants.
Section 4.1: Pairwise correlation of factors, Figure 5 and descriptive statistics, Table 1, ANOVA test and descriptive statistics.
Section 4.2: Figure 6 and descriptive statistics, Table 2.
Section 4.3: Artifact to artifact trust comparison, artifact to artifact acceptance rate comparison.
Section 5.1: Table 3.
Appendix B: Figure 9 and descriptive statistics, Table 4, Sybil trust plot and descriptive stats, Sybil trust modeling.

C.5.2 Qualitative Analysis

Execution Time: N/A

The qualitative results primarily report the counts of the coded qualitative data found in the file "qualitative_codings.xlsx". These code counts along with direct participant quotes inform Observations 5-10.
Section 5: Cohen's-Kappa - To find the interrating reliability of codes, we calculate Cohen's-Kappa for each codebook via the following script:

```
python3 codebook_kappa.py
```

Qualitative Reporting in Sections 5.1-5.3: For each of the following subsections, we note what findings were made, information was reported, and what specific sheet and cells (highlighted in colors) were used to inform these findings.
Section 5.1: Areas of Focus (sheet "factors_by_prompt"; highlighted in red), Artifacts Noticeability (sheet "factors_by_cond"; highlighted in red)
Section 5.2: Perception of Non-Existent Artifacts in Images (sheet "factors_by_cond"; highlighted in blue), Perception of Non-Existent Artifacts in Text (sheet "factors_by_cond"; highlighted in green)
Section 5.3: Noted UIs (sheet "strategies_by_prompt"; highlighted in red), Search for Personal Qualities (sheet "strategies_by_prompt"; highlighted in blue), Search for Inconsistencies (sheet "strategies_by_prompt"; highlighted in green), Reasons for Actions (sheet "strategies_by_prompt"; highlighted in orange).

C.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

Multiparty Private Set Operations is a software program that implements the protocols that we present in the main publication. The program enables multiple parties to privately compute the intersection of sets that they hold (MPSI) or the intersection of one set with the union of all other sets (MPSIU). If set elements have associated values, the library supports privately aggregating those values (MPSI-Sum or MPSIU-Sum). A delegated party learns the result of the set operation, and the parties learn no other information. The library implementation is in Go and supports execution in a Docker container.

A.2 Checklist

- **Algorithm:** Multiparty Private Set Operations implements the MPSI, MPSIU, MPSI-Sum, and MPSIU-Sum protocols in the main publication.
- **Compilation:** Compiling the program requires Go version 1.18 or more recent.
- **Data set:** The program generates random data to simulate protocol input in the user-specified `data` directory.
- **Metrics:** The program appends timing results to `bench.csv` in the user-specified `results` folder.
- **Output:** The program prints output to `stdout` and appends to `bench.csv`.
- **Experiments:** Please see the `README` file for guidance on replicating results in the main publication. The program reads protocol configuration parameters from `config.yml`.
- **How much disk space required (approximately)?:** Disk space requirements are proportional to the number of parties and set sizes, which are specified in `config.yml`.
- **How much time is needed to prepare workflow (approximately)?:** Both native and Docker builds take less than a minute on commodity hardware.
- **How much time is needed to complete experiments (approximately)?:** Please refer to Table 3 of the main publication.
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/citp/mps-operations
- **Code licenses (if publicly available)?:** We provide the program with the MIT License.
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/citp/mps-operations/releases/tag/usenix22

A.3 Description

A.3.1 How to access

Multiparty Private Set Operations is available in the Git repository at https://github.com/citp/mps-operations. The current version (as of publication) is https://github.com/citp/mps-operations/releases/tag/usenix22.

A.3.2 Software dependencies

Go (for native build) or Docker (for containerized build).

A.4 Installation

Native. Install Go (at least version 1.18) and run

```
go build -o mps_operations
./mps_operations
```

Docker. Install Docker (at least version 20.10.12) and run

```
docker build -t mps_operations .
docker run -it -rm -name mps_operations
mps_operations
```

A.5 Evaluation and expected results

Table 3 of the main publication provides execution times using large input set sizes. These benchmarks ran on a server using 128 cores. On personal computers, the execution times will be longer. In order to reproduce the benchmarks in Table 3, set the specified values for set sizes $|X_0|$ and $|X_i|$ in `config.yml`. Please refer to the `README` for build instructions.

A.6 Experiment customization

Please refer to `config.yml`.

A.7 Notes

- The number of parties n in `config.yml` does not include the delegated party.
- The program ignores the upper bound on associated values l in `config.yml` if the protocol is MPSI or MPSIU, because l is only necessary for value aggregation in MPSI-Sum and MPSIU-Sum.

A.8 Version

This artifact appendix is based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

Our artifact contains the implementation of constant-weight PIR as proposed in the paper titled “Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators".

We use this implementation to compare constant-weight PIR with other PIR protocols. We provide an implementation of folklore PIR for comparison. We provide scripts that use this implementation to generate the tables shown in the paper.

Aside from PIR, we provide scripts that benchmark the proposed equality operators and compare them with existing ones in terms of runtime.

A.2 Artifact check-list (meta-information)

- **Compilation:** The GNU GCC compiler (version >= 6.0) is required which supports OpenMP for parallelization. This compiler is publicly available.
- **Binary:** Binaries are not included but can be easily build using the steps outlined in the README. Two executables should be generated: `main` to experiment with PIR and `benchmark_eq` to benchmark the proposed equality operators.
- **Run-time environment:** Our code has been tested for Ubuntu 20.04. Besides the specified compiler, it requires the Microsoft SEAL library [1] to be installed.
- **Hardware:** Some runtimes in the paper are parallelized over 64 and 114 threads. To achieve the same results, it is required to have hardware with similar specs. The precise specs of the hardware are noted in the paper in each section.
- **Metrics:** In our PIR implementation, we measure the runtime of each step and the total runtime as well. We also measure the upload and download communication. In the benchmarks of equality operators, we measure the runtime.
- **Output:** The specified metrics are written to file (the name of the file is generated randomly) in the directory specified in the command line.
- **Experiments:** Scripts are provided to reproduce the results in the paper. These scripts run experiments and write the results to the ‘results’ directory.
- **How much disk space required (approximately)?:** All experiments are done in memory so not much disk space is required. However, the largest experiments use more than 100 GB of memory.
- **How much time is needed to prepare workflow (approximately)?:** Assuming all the prerequisites need to be installed, the installation time should not take more than 30 mins.
- **How much time is needed to complete experiments (approximately)?:** To reproduce all the results, the experiments take over a two weeks to run on a single machine. However, the number of runs can be reduced in all the provided scripts to reduce this to a couple days. Currently, the experiments are run 10 times.
- **Publicly available:** Our artifact is publicly available on Github at https://github.com/RasoulAM/constant-weight-pir
- **Code licenses:** The code is published under a BSD-3 license.
- **Archived (explicitly provide DOI or stable reference):** https://github.com/RasoulAM/constant-weight-pir/releases/tag/artifact-accepted

A.3 Description

A.3.1 How to access

The artifact is publicly available on Github at https://github.com/RasoulAM/constant-weight-pir/releases/tag/artifact-accepted.

A.3.2 Hardware dependencies

Some runtimes in the paper are parallelized over 64 and 114 threads. To achieve the same results, it is required to have hardware with similar specs. The precise specs of the hardware are noted in the paper in each section.

A.3.3 Software dependencies

This artifact runs on Ubuntu 20.04. It requires GNU GCC compiler (version >= 6.0) as the C++ compiler and the Microsoft SEAL library [2] to be installed. Instructions to install SEAL can be found in their repository.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

The instructions to build the repository are provided in the main README. The prerequisites such as the gcc compiler are specified in the README. Instructions on how to install the other dependencies such as SEAL and googletest are also specified.

[1] https://github.com/microsoft/SEAL

[2] https://github.com/microsoft/SEAL

A.5 Experiment workflow

We provide scripts to run the experiments outlined in the paper. These scripts are provided in the `src/build/scripts` directory. Details regarding these scripts and instructions on how to run them are given in `src/build/README.md`

To interpret the result of the experiments, we provide scripts in `src/build/interpret-results.ipynb`

A.6 Evaluation and expected results

We use this artifact to generate the tables shown in the paper, specifically Table 4, 5, 7 and 9 (and Table 12 and 13 in the appendix) . Instructions on which script to use to generate each table is given in `src/build/README.md`

A.7 Experiment customization

We provide a command line interface to experiment with different PIR protocols. Particularly, the user can experiment with folklore PIR and constant-weight PIR. All parameters can be assigned via the command-line. Instructions on what these parameters are and how to set them are given in the README. All results can be written to file and printed to the standard output.

Our scripts are not customizable (except for the number of runs) and are only used to automatically produce the results shown in the paper.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This implementation contains our incremental PIR protocol as well as two baseline PIR protocols described in the paper. Our implementation requires the dependencies specified in Section A.3. We did our experiments on CloudLab.

A.2 Artifact check-list

- **Compilation:** Follow the standard compilation steps in c++. We include the detailed instructions in `readme.md`.
- **Experiments:** See readme.
- **How much disk space required (approximately)?:** 2GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** 1 hour.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to access

See stable version at
`https://github.com/eniac/incpir/tree/a7d1bcf45b1bd5a3e98bcb421276ecd09c6eebdd`.

A.3.2 Hardware dependencies

Hardware should support AES-NI and AVX2.

A.3.3 Software dependencies

Protobuf, OpenSSL, libboost-all-dev, Python3, Matplotlib.

A.4 Installation

We provide a guide for how to install dependencies in `install.md`.

A.5 Experiment workflow

We provided scripts to generate numbers in benchmark table and graphs in the paper. See `readme.md` for more details.

A.6 Evaluation and expected results

We provide scripts to generate the data for the figures in the paper. In each folder (specified by `readme.md`), run `sh run.sh` or the instruction specified.

A.7 Notes

- The timings for some parts of the protocol could be slightly different from the results in the paper, because the client's queries are randomized and the time (for example, for Query) depends on which index it queries for. However, they should be on average close to what is shown in the paper.
- Some scripts will output "TimesNewRoman font not found" (if TimesNewRoman is not installed on the test machine), but you can still get figures without any problem.



A Artifact Appendix

A.1 Abstract

MIGP (Might I Get Pwned) is a next-generation password breach altering service to prevent users from picking passwords that are very similar to their prior leaked passwords; such credentials are vulnerable to credential tweaking attacks.

In summary, we are providing guidlines to evaluate the following results.

- [Figure 2]: Our proposed secure protocol for MIGP.
- Security simulation:
  - [Figure 8]: Simulation of attacker's success rate for different query budgets compared to traditional breach-altering service
  - [Figure 9]: Comparison of attack success rate for 'Das-R' and 'wEdit' for different query budgets.
- Performance simulaiton:
  - [Figure 12]: Average latency for different C3 services.
- Similarity simulation
  - [Figure 4]
  - [Figure 5]
  - [Figure 6]

A.2 Artifact check-list (meta-information)

- **Data set:** Since the files required to run the experiments are sensitive password leaks from 2019, if you need access to datasets please write to us. After downloading them, put the downloaded compressed file inside `path_to_MIGP/security_simulation/data_files` folder and then unzip it. For the `models.zip` file download it and put it inside the `similarity_simulation/artifact` folder.

  [Warning]. The zipped file is around 4.25 GB for the data files and 5.84 GB for the model files.
- **Software environment:** We have provided the required packages in `requirement.txt` file. We encourage the reviewers to use 'conda' or 'virtualenv' to create virtual environments and use pip to install them. We have used Python version 3.8.

  Before that, you will need to install the following three software packages.
  - `petlib` from here (For Figure 2, 12). Instructions are already in the link on how to install it.
  - Install argon2-cffi from here Installation - argon2-cffi 21.3.0 documentation.
  - GO (version 1.15) to run the WR19 and WR20 protocols in Figure 12. Make sure `GOPATH` variable points to `'path_to_MIGP_folder/performance_simulation/WR-19-20'`. To install GO version 1.15, we refer to the instructions from this link How To Install Go 1.14 on CentOS 8 | CentOS 7 | ComputingForGeeks. Additionally go the the `'path_to_MIGP_folder/performance_simulation/WR-19-20/src/` folder and run `'go get github.com/willf/bloom'`.
- **Hardware:** Our experiments were run on an Intel Xeon Linux machine with 56 cores and 125 GB of memory. You do not need any special hardware. But some security simulations may need large memory. Please let us know if you encounter such a memory error. We already provide the trained models.
- **Execution/compilation:** We have provided bash scripts to generate the figures. See section A.5.
- **Security, privacy, and ethical concerns:** Please DO NOT share this Google Drive link of the datasets with others as it contains leaked password datasets. Although these leaks are "publicly available", we request the reviewers to do so to safeguard against any problem. We also share the minimum version of the full leaked dataset that is required to evaluate the artifact. Moreover we request to delete the downloaded leaked dataset files after the evaluation is complete from the permanent storage.
- **How much disk space required: approximately** $\leq$ 13 GB

A.3 Description

A.3.1 How to access

Available at `https://github.com/islamazhar/MIGP_python/releases/tag/artifact_eval`. You can either clone or download the zipped source code.

A.3.2 Hardware dependencies

The security simulation for Figure 8 may require large memory. Please let us know if you encounter memory error while running those experiments.

A.3.3 Software dependencies

Already specified in A.2 software environment paragraph.

A.3.4 Data sets

Since the files required to run the experiments are sensitive password leaks from 2019, if you need access to datasets please write to us. We can grant access to datasets after properly reviewing the request.

A.3.5 Models

Already provided in the Google Drive link above.

A.3.6 Security, privacy, and ethical concerns

Already mentioned in A.2 in the "Security, privacy, and ethical concerns" paragraph.

A.4 Installation

Follow the “Software environment" paragraph in A.2.

A.5 Experiment workflow

A.5.1 Figure 2

- Expected time: 2-3 mins after installing the required software
- Required packages: petlib, argon2, all packages in requirements.txt
- Compilation: Go to ‘performance_simulation‘ folder and run the following commands.
  - In one terminal run the server using ‘`python3 MIGP_server.py`‘.
  - In another terminal run the query the server using ‘`python3.8 post_client_MIGP.py -username <username> -password <password>`‘
- How to evaluate: If you issue the following comands the expected outcome will be the following.

```
python3.8 MIGP_client.py --username Alice --
    ↪ password 123456 #will give exact
    ↪ password matching.
python3.8 MIGP_client.py --username Alice --
    ↪ password 123456$ # will give
    ↪ similar password matching
python3.8 MIGP_client.py--username Alice --
    ↪ password deercrossing # or any
    ↪ other password, will give not
    ↪ present in the leak
```

A.5.2 Figure 8

- Expected time: for budget $qc = 10, 100$‘. It takes less time but for $qc = 10^3$ expect 1-2 hours for $n = 10$ and 3-4 hours $n = 100$ depending on the memory and number of threads being run.
- Required packages: all packages in requirements.txt
- Compilation: We simulate the security simulation in three steps.
  - ‘`bash script_step_1.sh`‘. // This will create password variations. You can skip this one as we already provide the variations file inside ‘data_files‘ folder
  - ‘`bash script_step_2.sh`‘. // This create the top 10^3 guess ranks. We have also generated the guess ranks and balls of each password in the ‘data_files‘ folder. [skip if you want]
  - Finally, run "`bash script_step_3.sh <n> <qc>`" to generate the row corresponding to row with value ‘n‘ and ‘qc‘ in Figure 8. The results will be saved in ‘results/security_sumulation.tsv‘ file. This part may long time as for n =100 and $qc = 10^3$ it took us 12 hours to complete the simulation.
- How to evaluate: The results of each run will be saved at ‘results/security_simulation.tsv‘. Run ‘python3.8 Figure_9.py‘ to generate the Figure 8. If some values for ‘n‘ and ‘qc‘ the values has not been generated it will show blank.

A.5.3 Figure 9

- Expected time: 2-3 minutes
- Required packages: None
- Compilation: : Go to ‘`security_simulation/Figure_9` and run ‘python3.8 Figure 9.py‘
- How to evaluate: Inspect the generated ‘Figure_9.jpg‘ it should correspond to the paper presented in the paper (was just drawn using pgfplot for our paper)

A.5.4 Figure 12

We run the experiments on two EC2 instances as mentioned in the paper. But they can be tested on localhost as well. Make sure go (version 1.15) is installed and ‘GOPATH‘ points to path_to_MIGP_folder/performance_simulation/WR-19-20‘.

- Expected time: 1-2 hours. Basically, WR-19 and WR-20 take a lot of time.
- Required packages: specified in ‘requirements.txt‘ file
- Compilation:
  - In one terminal run the servers using ‘bash script_run_server.sh‘ and wait for some time for the servers to finish the precomputation.
  - On another terminal run ‘bash script.sh‘ and you will see the Figure on the terminal.
- How to evaluate: Since it is running on localhost the values may NOT exactly correspond to the values reported in the paper. But should follow a similar trend shown reported on the paper. Such As ‘IDB‘ protocols are the fastest ones. WR-19 and WR-20 are very expensive. MIGP_Hybrid should have low latency.

A.5.5 Figure 4,5,6

- Required packages: Training the Pass2Path models is computationally expensive. Therefore, we train these models in GPU and generated the prediction files for required test_files, to run the experiments fast. The code for training the Pass2Path models is in https://github.com/Bijeeta/credtweak/tree/master/credTweakAttack/. We also stored the sorted list of rules for Das-R and EDR models, ranked based on the breached dataset. Also make sure to install the packages mentioned in ‘requirements.txt‘
- Compilation: Go to “artifact” folder. Download the models.zip and copy the “models” folder here. Go to “artifact/src” and
  - For Fig-4, run "bash fig4.sh"
  - For Fig-5, run "bash fig5.sh"
  - For Fig-6, run "bash fig6.sh"
- How to evaluate: The values should match the ones in the figure.

A.6 Evaluation and expected results

Expected results are already mentioned for each of the Figures in A.5 section.

A.7 Notes

Please contact us via hotcrp if you face any problems or have any questions. Thanks for reviewing our artifact.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





D Artifact Appendix

D.1 Abstract

This artifact comprises several files that aid in the replication of our study: (1) a QSF-file containing all questions in a survey format exported from Qualtrics and that can be easily re-imported there); (2) a CSV-file with the the data collected from our participants with identifiable information removed (to improve compatibility, also a tab-separated version is provided); (3) the analysis script with the majority of the quantitative analyses of the paper; (4) a Jupyter Notebook file with the CHI-squared test; (5) the codebook of the qualitative analysis with counts for each of the codes. Using the data set and the analysis script, all quantitative results in the paper can be replicated.

D.2 Artifact check-list (meta-information)

- **Program:** The analysis was run with R version 4.2.0 running in RStudio[1] 2022.02.3 Build 492 with knitr. The following packages are needed to run the script: dplyr, AICcmodavg. For the chi-squared test, we used a Jupyter Notebook, version 6.4.8 to conduct the analysis. The easiest way to use Jupyter Notebook is to install Anaconda [2] which comes pre-installed with the most popular Python libraries and tools. Anaconda navigator version 2.2.0 as well as Python version 3.9.12 were used for this analysis. The following packages are required to run this script: pandas, numpy, scipy, statsmodels.
- **Compilation:** Some of the R packages and their dependencies require compilation, but R should handle this automatically when installing the packages.
- **Data set:** The data set collected from the participants of our study is included in the artifact
- **Run-time environment:** Recommended is use of RStudio 2022.02.3 Build 492 with R 4.2.0. Other configurations are likely to work but are untested. A Jupyter Notebook version 6.4.8 is recommended but not required to run the .ipynb. You can easily access the Jupyter Notebook by installing Anaconda. All our analyses were run on macOS.
- **Hardware:** No specific hardwre is needed.
- **Output:** The output on the R console and Jupyter Notebook represent the analyses as they were reported in the paper.
- **Experiments:** For a full replication of our study, the QSF-file can be used to import the survey back into qualtrics and distribute it among new participants. Note that the survey requires Javascript and therefore will not work with free Qualtrics accounts. For replication of the results reported in the paper, the analysis script and the data set collected from our participants should be used.
- **How much disk space required (approximately)?:** Negligible, less than 1MB.
- **How much time is needed to prepare workflow (approximately)?:** This depends on whether the required environment (RStudio and Anaconda) and the required packages are already installed. If none of the aforementioned are present, setup should take 30 minutes or less on a modern computer.
- **How much time is needed to complete experiments (approximately)?:** This depends on hardware, but should take less than 5 minutes on any recent laptop.
- **Publicly available (explicitly provide evolving version reference)?:** The artifact will be made available in a GitHub repo with a tag marking the version submitted for the artifact evaluation.
- **Code licenses (if publicly available)?:** The R code and Jupyter Notebook script are lincensed under the MIT license.
- **Data licenses (if publicly available)?:** The data is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.[3]
- **Archived (explicitly provide DOI or stable reference)?:** GitHub Commit ID for the version submitted for artifact evaluation: 2ead79bbe026789bc78d87b420c487da4d980ff5
  GitHub Commit ID for the version improved with the comments from artifact evaluation reviews:
  a90e474e2e2be23898b4b85570cd0daaba61970f

D.3 Description

D.3.1 How to access

The artifact can be downloaded from the GitHub repository.[4]

D.3.2 Hardware dependencies

N/A

D.3.3 Software dependencies

The analysis requires R to run. Recommended is RStudio 2022.02.3 Build 492 with R 4.2.0, since the authors used these versions. Other versions are likely to work but are untested. RStudio can be obtained for free online.[5] The following R packages are needed to run the script: dplyr, AICcmodavg. For the chi-squared script, Jupyter Notebook version 6.4.8 is recommended but not required. To quickly use Jupyter Notebook, download Anaconda. [6]

D.3.4 Data sets

No third-party data sets were used.

D.3.5 Models

N/A

[1] https://www.rstudio.com/products/rstudio/download/
[2] https://www.anaconda.com/products/distribution
[3] http://creativecommons.org/licenses/by-nc-nd/4.0/
[4] https://github.com/gwusec/2022-USENIX-Password-Managers
[5] https://www.rstudio.com/products/rstudio/download/
[6] https://www.anaconda.com/products/distribution

D.3.6 Security, privacy, and ethical concerns

N/A

D.4 Installation

The setup consists of two steps. First, R needs to be installed. Recommended is RStudio 2022.02.3 Build 492 with R 4.2.0, since the authors used these versions. Other versions are likely to work but are untested. RStudio can be obtained for free online.[7] After the installation of RStudio, R should be available as well.

When RStudio is installed it must be started and the analysis script can be opened using the `File` dialog. Then the following R packages need to be installed: dplyr, AICcmodavg. To install these packages using RStudio, open the `Tools` menu and then select `Install packages...`. In the search box enter the first package. Then click `install`. Repeat these two steps for the second package. Installation of the packages might take some time if they need to be compiled. Once the two packages are installed, the analysis script can be run.

To run the .ipynb stats script that has the chi-squared test, first ensure you have Python installed as well as all the required dependencies. Python version 3.9.12 was used for this analysis. In addition to Python, the following dependencies also need to be installed: pandas, numpy, scipy, statsmodels. You can install them one by one from the terminal using pip (which is automatically installed with Python):

```
pip install pandas
pip install numpy
pip install scipy
pip install statsmodels
```

Once Python and all the above dependenices have been installed, you will be ready to run the Jupyter Notebook script. It is recommended you download Anaconda which comes pre-installed with Jupyter Notebook. Anaconda can be obtained for free online.[8] Once Anaconda is installed, open it and launch Jupyter Notebook, and browse to the location of the script. Run all the cells, one by one from top to bottom. It should print the results to the screen.

D.5 Experiment workflow

The R analysis script is divided into several segments called "chunks", each pertaining to the preparation of a specific variable or performing a specific analysis. These chunks are delimited by three accents before and after the block. Each chunk is labeled. The label is enclosed by curly brackets. The respective syntax looks like this:

```
```{r <section label>}
    <R code>
```
```

The easiest way to run the analyses is to run the script chunk-by-chunk from the top in RStudio. Running a chunk can be achieved in RStudio in three ways. Firstly, RStudio provides a small green right-arrow button on the top right for each chunk. Clicking it will run the respective chunk. Secondly, with the curser in a chunk, you can use the shortcut `Ctrl + Shift + Enter` (on macOS: `Cmd + Shift + Enter`) to run the respective chunk. Thirdly, with the curser in a chunk, you can use the menu `Code` → `Run Region` → `Run Current Chunk`.

The Jupyter Notebook script is similarly divided into several cells. Run the cells one by one, from the top to the bottom and the results will be displayed on the screen.

D.6 Evaluation and expected results

Claim 1: Awareness and use of PMs is much broader than previously reported The overall high awareness and use of password managers are supported by the analyses in section "prepare pwdm awareness variable" and "prepare password manager use variable" respetively.

Claim 2: The vast majority of respondents reuse passwords across accounts The results pertainig to password reuse can be found in section "RQ-2 reuse."

Claim 3: Perceived ease-of-use overall plays a key role in password manager adoption The results for the regression analysis identifying ease-of-use as predictor when all PM-users are considered can be found in section "pwdm use." The regression analyses for only browser-based password managers, system password managers, and third-party password managers can be found in the sections "browser pwdm use," "system pwdm use," and "third-party pwdm use" respectively.

Claim 4: Third-party password manager users are significantly more likely to use the PM to generate passwords The results for this can be replicated by running the Jupyter Notebook file called chi_test.ipynb. These will be printed to the screen.

Claim 5: The majority of participants would adopt a PM if it was offered to them for free by their organization The analysis pertaining to the adoption of password managers when one is offered by the participant's organization can be found in section "prepare pwdm use in organization variable."

D.7 Experiment customization

N/A

D.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

[7] https://www.rstudio.com/products/rstudio/download/
[8] https://www.anaconda.com/products/distribution



A Artifact Appendix

A.1 Abstract

Our artifact contains the source files, scripts, and other necessary files for reproducing the results described in the paper. It consists of the two compilers (vWasm and rWasm), the semantics fuzzer, as well as benchmarking scripts. To run these, one needs a Linux x86-64 machine, or a Docker environment capable of running it. Since one of the two compilers, namely vWasm, contains a machine-checked proof, the artifact also contains instructions to re-verify that all parts of the proof are indeed accepted by F*.

A.2 Artifact check-list (meta-information)

- **Program:**
  - vWasm: a formally-verified provably-safe sandboxing compiler, built in F*
  - rWasm: a high-performance informally-verified provably-safe sandboxing compiler
  - wasm-semantics-fuzzer: a tool for providing greater assurance in the semantic correctness of any Wasm implementation
- **Compilation:** vWasm requires F*, OCaml, nasm, etc.; the rest require a Rust installation. We include a Docker image with all requirements in the artifact.
- **Data set:** Benchmarks and micro-benchmarks are included in the artifact. See `benchmarks/`, `microbenchmark-compare-read-arr/` and `image-conversion-scenario/` in the main repository.
- **Run-time environment:** Our artifact was developed and tested on recent Linux-based systems. We include a Docker image with all requirements.
- **Hardware:** Requires an x86-64 machine. We tested on an AMD Ryzen 3700x (64GB memory) and on an Intel i9-9900K (128GB memory). While almost everything *should* run on a machine with less memory, we recommend 32GB or higher to allow parallelism to save user time in some of the memory-intensive steps.
- **Output:** We provide more detail in the artifact `README.md` files, but in short, building vWasm will verify and compile the vWasm compiler, building rWasm will compile the rWasm compiler, building wasm-semantics-fuzzer will build the fuzzer, and running the benchmarks will use the compilers to run the experiments described in the paper.
- **How much disk space required (approximately)?:** Approximately 5 GB for the Docker image; the rest of the files are negligible in size. When running benchmarks, space usage can increase a lot more, and thus it is best to have free space on the order of approximately 100 GB.
- **How much time is needed to prepare workflow (approximately)?:** The provided Docker image contains all requirements for the two compilers, and loading it from the exported image should only take a minute or two. Building the Docker from scratch takes significantly longer (an hour or two). The other Wasm runtimes being benchmarked against are not all included in the Docker image, but instructions are included and should not take more than 15 minutes to get running.
- **How much time is needed to complete experiments (approximately)?:** Re-verifying vWasm, and running the main execution-time benchmarks are the most time consuming parts of the artifact. In total, expect this to take multiple hours, with times varying depending on the available parallelism on the machine being tested on.
- **Publicly available:** The latest version of the repositories:
  - https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22 (top-level repository that contains the benchmarks, and imports the rest as git submodules)
  - https://github.com/secure-foundations/vWasm
  - https://github.com/secure-foundations/rWasm
  - https://github.com/secure-foundations/wasm-semantics-fuzzer

  The first of the above links contains the other three as git submodules, pinned to specific git commits.
- **Code licenses:** BSD 3-Clause License
- **Archived (stable reference):** Top-level repository, with the other repositories fixed to specific git commit hashes: https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22/tree/6f5668d3f216aeef65cf2bf2d916a40d3c750e53

A.3 Description

A.3.1 How to access

https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22 is a link to the top-level artifact repository, which contains the

rest of the related repositories as git submodules. To get them all in one single command, run `git clone --recursive https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22`

Instructions for obtaining the vWasm Docker image can be found at https://github.com/secure-foundations/vWasm/tree/main/.docker, and a top-level `Dockerfile` can be found at the root of the top-level repository.

A.3.2 Hardware dependencies

Requires an x86-64 machine. 32+ GB of memory is recommended.

A.3.3 Software dependencies

Requires Docker installed, preferably on a Linux host. All other requirements work inside the container.

A.3.4 Data sets

Included in the artifact.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

We provide detailed instructions throughout the artifact in the form of `README.md` files.

In short, install Docker (https://www.docker.com/get-started), recursive-clone the repositories (`git clone --recursive ...` command from above), go to `provably-safe-sandboxing-wasm-usenix22/vWasm/.docker` and follow instructions there to download and import the pre-built image, and then jump inside the provided Docker container. Following this, everything else can be run inside the container.

A.5 Experiment workflow

For each experiment, we provide a relevant `README.md` file with detailed instructions. We recommend executing steps in the following order:

1. Build the Docker image, and jump into the container.
2. Run the verification and build process for vWasm.
3. Run the build process for rWasm.
4. Generate `.wasm` files using wasm-semantics-fuzzer and run them in vWasm and rWasm.
5. Run the microbenchmark.
6. Run the execution-time benchmarks in the `benchmarks/` directory (here, you can choose to compare against all other tools, or you can run fewer tools, see the `README.md` for instructions).
7. Run the image-conversion-scenario.

A.6 Evaluation and expected results

Our paper claims to make the following contributions (copied verbatim from Section 1):

1. An exploration of two distinct techniques to achieve provably safe, performant, multi-lingual sandboxing. We implement these as open-source tools, and evaluate them on a collection of quantitative and qualitative metrics.
2. vWasm, the first verified sandboxing compiler for Wasm, achieved via traditional machine-checked proofs.
3. rWasm, the first provably safe sandboxing compiler with competitive run-time performance. We achieve this using non-traditional repurposing of existing tools to provide provable guarantees without writing formal proofs.

We provide detailed instructions throughout the artifact in the form of `README.md` files.

To confirm that vWasm is formally verified, execute the steps in `vWasm/README.md`. Each file in the project is machine-checked and only once all files are verified by F*, will it produce the extracted OCaml files, which are then compiled to an executable compiler. The high-level theorem statement being proven can be found in `vWasm/compiler/sandbox/Compiler.Sandbox.fsti`.

Both vWasm and rWasm can be run independently to compile any Wasm module. Built-in runtime support is provided for Wasm modules that expect a WASI interface.

Using wasm-semantics-fuzzer, one can perform validation checks that the semantics implemented by vWasm and rWasm do indeed match expected Wasm semantics.

The image-conversion-scenario demonstrates a converter from GIFs to JPEGs, using version of libraries susceptible to CVE-2008-0554. Using `./see_cve_impacts.sh`, one can run a proof-of-concept input that demonstrates how the native, vWasm-built, and rWasm-built versions of the programs perform. Expected behavior is detailed further in the script before each execution, but in summary, the native version (not protected by vWasm or rWasm) will suffer a bad crash, while the vWasm and rWasm versions will successfully safely trap the violation.

Quantitative evaluation is performed using the benchmarks and provided scripts. For execution time and run time benchmarks, results should be within the error bars, if run on a similarly modern hardware. The microbenchmarks are more susceptible choice of hardware (as shown in Figure 8), and we have tested only on an AMD Ryzen 3700x and an i9-9900k.

A.7 Experiment customization

After vWasm and rWasm are built, you can test them with any WASI-enabled modules you like. Instructions are provided in the relevant `README.md` files.

For the execution-time benchmarks, not all competing Wasm execution runtimes are included in the vWasm Dockerfile, but the top-level Dockerfile does include them all. Additionally instructions for how to install them, or how to selectively disable the runtimes are given.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

This artifact is provided to help validate two goals of our proposed platform SWAPP: compatibility (changes needed to work with legacy code or other existing libraries); and fast-prototyping (easiness to program a new app and its effectiveness). Consequently, the artifact contains two major components corresponding to each goal. First, we provide clean SWAPP and its app source codes. This will be used in conjunction with Wordpress and Workbox to show how to encapsulate Workbox as a SWAPP app and run SWAPP in a popular web app (Wordpress) as discussed in Section 6.2. Second, we provide four demo (pre-configured SWAPP and its apps) that illustrates how four of the apps discussed in the paper can work to prevent the corresponding attacks. To run this artifact, we provide Docker images with shell scripts that will help set up the environment automatically.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Ubuntu 18.04+ and Docker.
- **Metrics:** Compatibility with legacy code. Vulnerabilities mitigated.
- **Output:** Web page. Console. Measured characteristics.
- **Experiments:** Manual steps by users.
- **How much disk space required (approximately)?:** 1GB.
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes.
- **How much time is needed to complete experiments (approximately)?:** 30 minutes
- **Publicly available (explicitly provide evolving version reference)?:** Yes. https://github.com/successlab/swapp
- **Archived (explicitly provide DOI or stable reference)?:** Yes. https://doi.org/10.5281/zenodo.6860277

A.3 Description

A.3.1 How to access

SWAPP is publicly available at https://github.com/successlab/swapp. The artifact is available at https://doi.org/10.5281/zenodo.6860277.

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

Ubuntu 18.04+. Docker.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

We have provide docker images with two shell scripts to help install Docker (install.sh) and setup the environment (deploy.sh). Users only need to execute these scripts in an Ubuntu system as required.

A.5 Experiment workflow

There are two metrics to validate our artifact: compatibility (M1), and programmability (M2). The workflow of this experiment is split into two sections correspondingly.

Section M1 showcases the compatibility of SWAPP. There are two steps in this section.

1. Setup Wordpress. Simply visit http://localhost using a web browser and follow the page instruction.
2. Interact with SWAPP. The installed Wordpress is already equipped with SWAPP. Four apps are also enabled. Interact with the website and see the browser console to observe the interaction and performance of SWAPP.

Section M2 showcases the programmability of SWAPP. We provide four demonstrating web pages corresponding to each of the four apps discussed in the paper: DOM Guard, Cache Guard, Autofill Guard, and Data Guard. The demo should also illustrate the effectiveness of each apps in responding to the corresponding attacks.

DOM Guard's effectiveness in preventing DOM-XSS attacks can be observed. Visit http://localhost/demo/domguard/index.html using a web browser to access DOM Guard's demo web page. Further instructions are provided in the web page.

Cache Guard's effectiveness in preventing side-channel attacks can be observed. To validate Cache Guard, simply visit http://localhost/demo/cacheguard/index.html using a web browser. Further instructions are provided in the web page.

Autofill Guard's effectiveness in preventing XSS attackers from accessing user's form input can be observed. Visit http://localhost/demo/autofillguard/ using a web browser to access Autofill Guard's demo web page. The website is installed with phpBB and the following credentials need to be used to correctly set up the demo.

- Database server hostname: mysql
- Database username: wp_user
- Database password: wp_password
- Database name: wordpress

After the set up is done, remove the /public_html/demo/autofillguard/install folder. Then, click "Take me to the ACP" and click "Logout" of the admin account. Next, revisit http://localhost/demo/autofillguard/. There should be a login form within an iFrame. In the case the iFrame does not show up, try refreshing the web page. Interact with the login form using the admin credentials to see if Autofill Guard works.

Data Guard's effectiveness in preventing Indirect Object Reference attacks can be observed. To validate Data Guard, simply visit http://localhost/demo/data_guard/index.html and follow the instruction given in the web page.

A.6 Evaluation and expected results

There are two goals of SWAPP that this artifact aims to validate.

SWAPP requires minimal changes to legacy and existing code (Compatibility). In section M1 workflow, we demonstrate that SWAPP can be easily installed on a popular web app like Wordpress. For instance, SWAPP only requires one line of code change to work with WordPress. Furthermore, encapsulating Workbox, a popular caching library, as a SWAPP app only requires a few lines of code change. The specific files that we change are located at public_html/wp-content/themes/twentytwentyone/footer.php (line 13) and public_html/apps/workbox-sw.js (lines 88-124). By observing the console while using Wordpress, there should not be any fatal errors from SWAPP.

SWAPP apps can be easily developed and are effective (Fast-prototyping). In section M2 workflow, we provide several demonstrating web pages for testing four SWAPP apps. Interacting with the demo should show that SWAPP and the apps are effective.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

Our artifact is the code used for our data collection and the analysis of the collected data. For the results of our paper, we focus on the data of our first crawl (2nd January 2022). We executed the scripts on a machine with 4 Intel(R) Xeon(R) Platinum 8160 CPUs (192 Cores total), 1.5 TB of RAM, and a one Gbps network connection. This execution took less than two days but will take longer if a machine has less CPU Power or a slower network connection. We provide a docker-compose config file to execute our pipeline inside a docker container regarding the software requirements. In this case, any machine that can build and spawn docker containers should work. We also provide files, installation scripts, and requirements.txt files for direct execution on a Linux system. In the appendix of our paper, we also listed other crawls and their overlaps between the first and follow-up crawls (Paper Appendix B). The results of the follow-up crawls, and thus also the results of the artifact execution, highlight that our results can be confirmed over multiple crawls. The expected results of the execution of our artifact is a table similar to Table 4 of our paper. Notably, the results might vary slightly (see Paper Appendix B), especially now the numbers might be lower due to our notification campaign (see Paper Section 6.4 Disclosure).

A.2 Artifact check-list (meta-information)

- **Data set:** Tranco List from 01-01-2022
- **Hardware:** In general no restrictions, but depending on the CPU Power and Network speed it will take longer.
- **Security, privacy, and ethical concerns:** Crawl process might put load on the crawled servers.
- **Metrics:** Number of sites that have inconsistent security configurations.
- **Output:** The script will print on console output individual numbers as well as a latex table similar to the paper's Table 4.
- **Experiments:** Results might vary slightly (see Paper Appendix B), especially now due to our notification campaign.
- **How much disk space required (approximately)?:** ~80GB
- **How much time is needed to prepare workflow (approximately)?:** Docker Setup: 5-10 min; Manual Setup: 10-15 min.
- **How much time is needed to complete experiments (approximately)?:** Highly depends on CPU/Network speed, for us the crawl took less than two days.
- **Publicly available (explicitly provide evolving version reference)?:** GitHub Repository incl. version history[1]
- **Code licenses (if publicly available)?:** AGPL-3.0 license
- **Archived (explicitly provide DOI or stable reference)?:** Stable reference to Git Commit[2].

A.3 Description

A.3.1 How to access

We made our pipeline publicly available via GitHub[1]. The stable reference for the submitted version is the commit where we incorporated the feedback of our reviewers[2].

A.3.2 Hardware dependencies

In general no restrictions, except ~80GB free space and enough CPU Power and RAM to perform HTTP requests. We executed the scripts on a machine with 4 Intel(R) Xeon(R) Platinum 8160 CPUs (192 Cores total), 1.5 TB of RAM, and a one Gbps network connection. The execution took us less than two days but will take longer if a machine has less CPU Power or a slower network connection.

A.3.3 Software dependencies

We provide a docker-compose config file to execute our pipeline inside a docker container regarding the software requirements. In this case, any machine that can build and spawn docker containers should work. We also provide files, installation scripts, and requirements.txt files for direct execution on a Linux system. The installation script will install `python3`, `pip`, `curl`, `tor`, `openvpn`, `psmisc`, `wget`, and `fping`. For the execution of the python scripts the *requirements.txt* contains `requests`, `tldextract`, `psycopg2`, `beautifulsoup4`, `lxml`, and `pysocks`.

A.3.4 Data sets

As a list of sites where we want to investigate if they have inconsistent behavior we used the Tranco List of Top sites from 01-01-2022 (see `scripts/xvwn_20220101.csv` in our repository).

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

Not too many of the crawls should be conducted in parallel because it might put load on the crawled Web sites, which might interfere with their availability or response speed.

A.4 Installation

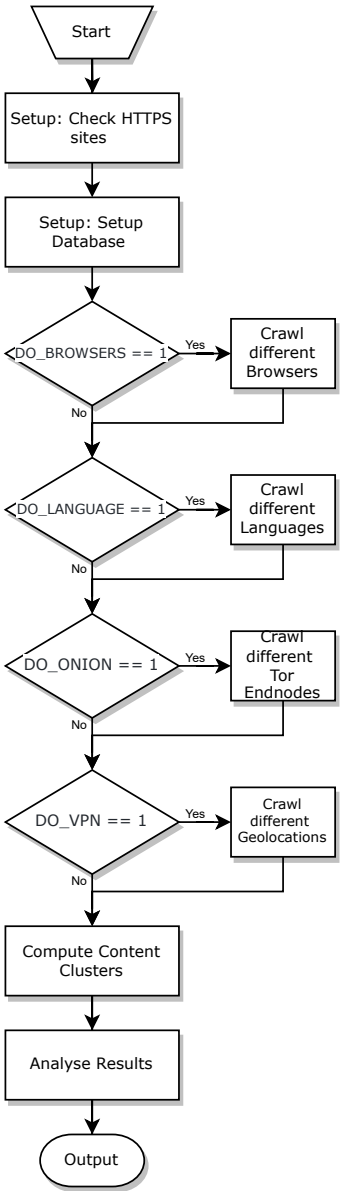
For the **docker** way you need to install the `docker`, `docker-compose`, and if you want to also crawl VPNs the `openvpn` package. Afterwards you can configure the crawl by editing the `docker-compose.yaml` file according to the GitHub README.md. Then you can build the docker via executing `docker-compose build` in the root folder of our repository. For **manual** setup you can execute `install.sh` which installs all

[1] https://github.com/cispa/the-security-lottery

[2] https://github.com/cispa/the-security-lottery/tree/66cc012fe7603e1758dde68fe9eec2d23542968e

dependencies. Then you should also set your environment variables to the values that are specified within the `environment` part of the `docker-compose.yaml` file. Afterwards you can execute the pipeline by executing `start.sh` or following the steps specified in the Repo's README.md.

A.5 Experiment workflow



A.6 Evaluation and expected results

Our measurement shows that a significant fraction of the Top Web sites suffer from different types of client-side security inconsistencies. Remarkably, the inconsistencies of 194 sites can be attributed to specific client characteristics, which identify weak spots in the security configuration, while the inconsistencies of 127 other sites can be attributed to non-deterministic factors, which may nevertheless be exploitable by an attacker.

Detailed numbers for the detected intra-test and inter-test inconsistencies for each factor and security mechanism are presented in Table 4 of the paper. Here we also present the numbers with and without page similarity for HSTS to highlight the impact of this choice on the measurement. The numbers in this table are also the output of our main analysis script (`scripts/sql_table.py`), which is executed as the last part of our pipeline. Furthermore details about the specific inconsistencies are saved into the `/data` directory as JSON files. Notably, the current numbers might be lower due to our notification campaign (see Paper Section 6.4 Disclosure), where we notified the affected parties about the problem and got responses that the issue had been fixed. Also, due to the nature of some of the inconsistencies, especially the intra-test inconsistencies, the number in general, varies as we depicted in Appendix B of the paper.

A.7 Experiment customization

One can specify which of the crawls should be performed by changing the corresponding DO_<BROWSER|LANGUAGE|ONION|VPN> values to 1=enabled or 0=disabled. Notably, when you want to execute the crawls individually, you need to set SKIP_SETUP to 1 after the first crawl, such that the database is not cleared. Also, note that the VPN crawl requires valid hidemyass.com credentials.

A.8 Notes

The results might vary slightly from ours due to the nature of inconsistencies (see Paper Appendix B). Especially now, the numbers might be lower due to our notification campaign (see Paper Section 6.4 Disclosure), where we notified the affected parties about the problem and got responses that the issue had been fixed.

Also, note that the VPN crawl requires valid hidemyass.com credentials in order to work.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact is based on PyTorch and requires GPU support. We implemented Algorithm 1 (double-masking algorithm) and Algorithm 2 (robustness certification procedure) of our PatchCleanser paper. The artifact can reproduce all experimental results (*clean accuracy and certified robust accuracy*) reported in the main body of the paper.

Our source code is available at https://github.com/inspire-group/PatchCleanser/tree/2370c78da15ccec08b7a05145c92cafb9b0f73a9. We further provide a detailed guide for evaluating our artifact at https://github.com/inspire-group/PatchCleanser/blob/2370c78da15ccec08b7a05145c92cafb9b0f73a9/misc/reproducibility.md.

A.2 Artifact check-list (meta-information)

- **Algorithm:** We implement Algorithm 1 (double-masking algorithm) and Algorithm 2 (robustness certification procedure) of our paper.
- **Program:** N/A.
- **Compilation:** N/A.
- **Transformations:** N/A.
- **Binary:** N/A.
- **Model:** ResNet, Vision Transformer, ResMLP. We provide downloading links to pretrained weights.
- **Data set:** ImageNet, ImageNette, CIFAR-10. They are publicly available benchmark datasets.
- **Run-time environment:** We test our artifact using anaconda virtual environment on Linux.
- **Hardware:** Requires one GPU.
- **Run-time state:** N/A.
- **Execution:** N/A.
- **Security, privacy, and ethical concerns:** N/A.
- **Metrics:** (top1) clean accuracy and (top1) certified robust accuracy. They are defined in the paper (at the end of Section 4.1).
- **Output:** We output results to console; they are numerical accuracy values.
- **Experiments:** We provide scripts for running experiments.
- **How much disk space required (approximately)?:** Datasets takes around 7GB. Each model weight take 100-300MB. The mid-product of the experiments takes less 300 MB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** Each experiment takes 3 hours to 2 days (from scratch).
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/inspire-group/PatchCleanser.
- **Code licenses (if publicly available)?:** MIT License.
- **Data licenses (if publicly available)?:** N/A.
- **Workflow frameworks used?:** N/A.
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/inspire-group/PatchCleanser/tree/2370c78da15ccec08b7a05145c92cafb9b0f73a9.

A.3 Description

A.3.1 How to access

We host our source code on GitHub at https://github.com/inspire-group/PatchCleanser.

Specifically, we use this commit for the artifact evaluation: https://github.com/inspire-group/PatchCleanser/tree/2370c78da15ccec08b7a05145c92cafb9b0f73a9.

A.3.2 Hardware dependencies

The artifact requires 2 CPU cores and 1 GPU.

A.3.3 Software dependencies

The artifact is based on Python, PyTorch, `timm`, and other Python packages. All packages can be easily installed with pip; we provide a list of required packages in `requirement.txt`.

A.3.4 Data sets

We use three publicly available datasets in our evaluation: ImageNet, ImageNette, CIFAR-10. See our reproducing instructions for more details.

A.3.5 Models

We use three representative image classifier models: ResNet, ResMLP, and Vision Transformer. We build models using `timm`; we provide download links to our pretrained weights. See our reproducing instructions for more details.

A.3.6 Security, privacy, and ethical concerns

N/A.

A.4 Installation

1. Install Python. [help link]
2. Install GPU-compatible PyTorch. [help link]
3. Install other Python dependencies. [help link]

4. Clone the source code from https://github.com/inspire-group/PatchCleanser/tree/2370c78da15ccec08b7a05145c92cafb9b0f73a9

5. Download datasets. [help link]

6. Download pretrained weights. [help link]

A.5 Experiment workflow

Our experiment is mostly based on the script `pc_certification.py`, in which we implemented our double-masking algorithm (Algorithm 1) and certification procedure (Algorithm 2). By running this script with proper command (we provide all necessary shell commands to run `pc_certification.py` in our reproducing instructions), we can read clean accuracy and certified robust accuracy from the console. We can compare the obtained results with the results reported in the paper to validate the reproducibility of our paper.

A.6 Evaluation and expected results

Our main claim is that our PatchCleanser defense achieves state-of-the-art defense performance, in terms of certified robust accuracy and clean accuracy, against adversarial patch attacks. This claim is supported by Table 2 of our paper. We can use commands listed in this section of our reproducing instructions to generate our key results, which should match the results reported in Table 2 of our paper.

In addition to our key results, our artifact also supports other experimental analyses presented in Section 4 and Section 5 of our paper. We provide detailed instructions and shell commands in our reproducing instructions.

Our algorithms are deterministic. Therefore, we do not expect any large variation in results. However, it is possible to have tiny mismatches ($< 0.1\%$) due to the imprecise float point computation on different hardware (e.g., GPUs).

A.7 Experiment customization

Our source code provides an easy way to customize the experiment.

First, we already support three datasets (ImageNet. ImageNette, CIFAR-10) in this artifact. In our GitHub repository, we further support three additional datasets (CIFAR-100, SVHN, Flowers-102). We can also add other datasets if their pretrained weights are available; we only need to register the new datasets in `utils/setup.py`.

Second, we can also support other image classification models (other than ResNet, ResMLP, ViT). We only need to have pretrained weights for these models and register them in `utils/setup.py`.

Third, we can easily change the parameters of our system. In our source code, we can change the underlying classification model using the flag `-model`, change the number of masks using `-num_mask`, change the mask stride using `-mask_stride`, change the estimated patch sizes using flags `-patch_size`, `-pa`, and `-pb`. We have already evaluated the effect of different parameters in our paper and this artifact.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact includes the code necessary to reproduce the experimental results presented in our paper titled "Transferring Adversarial Robustness Through Robust Representation Matching". It is made available in the form of a GitHub repository (final stable URL: https://github.com/Ethos-lab/robust-representation-matching/releases/tag/final). Our experiments involve training neural network based image classifiers that are robust against adversarial attacks. Therefore, we provide the necessary training and evaluation scripts, along with all the supporting code. The expected results, as reported in the paper, are : (1) total training time, and (2) accuracy of the trained classifier on clean and adversarial test sets. All our code is written in Python.

On the hardware side, the code requires a machine with at least one GPU with 12 GB memory and storage space > 150 GB to run. We recommend at least 8 GB of RAM. On the software side, the code requires Python compiler, pip, conda and several other 3rd party Python libraries like PyTorch, IBM's Adversarial Robustness Toolbox, Nvidia's apex *etc.* Detailed instructions regarding setting up the run-time environment are provided in Section A.4 and the repository README.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Our paper presents a novel algorithm called Robust Representation Matching (RRM). The purpose of this algorithm is to speed up the process of adversarially training neural network based image classifiers.
- **Data set:** We perform experiments using two image datasets: CIFAR-10 and Restricted-ImageNet. The CIFAR-10 dataset downloads itself if not available. It requires 341MB storage space. For experiments involving Restricted-ImageNet, the full ImageNet dataset needs to be downloaded. Instructions for this are provided in the README of the code repository. It requires 145 GB storage space.
- **Model:** The CIFAR-10 experiments are conducted using the following neural networks: VGG11, VGG19, ResNet18, ResNet50. The Restricted-ImageNet experiments use the following neural networks: AlexNet, VGG16, ResNet50. All the code associated with these networks is provided in the repository. We also make available weights of pre-trained classifiers for quick evaluation.
- **Run-time environment:** Our code has been tested on a Linux machine. To prepare the run-time environment, one needs to create a Python virtual environment and install all required Python libraries. The instructions for setting up the run-time environment are provided in Section A.4 and the README in the repository.
- **Hardware:** The code requires a machine with at least one GPU with 12 GB memory and storage space > 150 GB. We recommend running the Restricted-ImageNet training scripts on 4 GPUs. Also, we recommend 8 GB of RAM.
- **Execution:** Here we provide estimated time taken by different components of our experiments. These estimates were computed on our machine. We ran our experiments on two different machines. The CIFAR-10 experiments were run on a machine with an Intel Xenon(R) Gold 6136 CPU, 16 GB RAM, and an Nvidia Titan V GPU. The training scripts took $\sim$ 5 hours on average. In total 19 classifiers need to be trained using different methods. The Restricted-ImageNet experiments were run on a second machine with an Intel Xenon(R) E5-2690 CPU, 16 GB RAM, and an Nvidia V100 GPU. The training scripts took $\sim$ 1 week to run on average. In total 5 classifiers need to be trained using different methods. For both the datasets, the evaluation scripts take $\sim$ 3 hours to run in the worst case, with every trained model needing to be evaluated once.
- **Metrics:** We report two metrics in our paper: (1) Training run time and (2) Accuracy of clean and adversarial test sets. Note that due to differences in hardware, the absolute training times will be different than what is reported in the main paper. However, the speedup (ratio of train times) should be approximately the same.
- **Output:** All the expected output will be printed out as stdout on running the evaluation script. The following quantities will be outputted: (1) average time per training epoch, (2) its 95% confidence interval, (3) total training time, (4) accuracy on clean test set, and (5) accuracy on adversarial test set.
- **Experiments:** The step-by-step instructions to reproduce the experimental results are provided in the GitHub READMEs. The accuracy numbers will be within a few percentage points of the numbers reported in the main paper. The absolute training time numbers will vary from what is reported in the main paper due to hardware differences. However, the speedup numbers (ratio of training times) will be approximately the same.
- **How much disk space required (approximately)?:** 150 GB
- **How much time is needed to complete experiments (approximately)?:** On our machine, training all the reported classifiers on CIFAR-10 took $\sim$ 4 days. Training all the reported Restricted-ImageNet classifiers took $\sim$ 5 weeks. Evaluating all the classifiers (corresponding to both datasets) took $\sim$ 3 days. During reproduction, expect significant variations in these times because of hardware differences.
- **Publicly available (explicitly provide evolving version reference)?:** Yes. https://github.com/Ethos-lab/robust-representation-matching
- **Code licenses (if publicly available)?:** MIT License
- **Data licenses (if publicly available)?:** CIFAR-10: no license. ImageNet: https://www.image-net.org/download.php.

A.3 Description

A.3.1 How to access

Clone GitHub repository, available here (final stable URL): https://github.com/Ethos-lab/robust-representation-matching/releases/tag/final

A.3.2 Hardware dependencies

The code requires a machine with at least one GPU with 12 GB memory and storage space > 150 GB. We recommend running the Restricted-ImageNet training scripts on 4 GPUs. Also, we recommend 8 GB of RAM.

A.3.3 Software dependencies

Our code is written in Python and requires a Python compiler installed along with the python package managers pip and conda. In addition, our code makes use of several 3rd party Python libraries. For instructions regarding how to install all the software dependencies and set up the run-time environment, refer to Section A.4 and the GitHub README.

A.3.4 Data sets

We use two datasets in our experiments: CIFAR-10 and Restricted-ImageNet. CIFAR-10 will download itself if not available. For Restricted-ImageNet, the entire ImageNet dataset needs to be downloaded. Instructions for this are provided in the GitHub README.

A.3.5 Models

The CIFAR-10 experiments are conducted using the following neural networks: VGG11, VGG19, ResNet18, ResNet50. The Restricted-ImageNet experiments use the following neural networks: AlexNet, VGG16, ResNet50. All the code associated with these networks is provided in the repository. We also make available weights of pre-trained classifiers for quick evaluation.

A.3.6 Security, privacy, and ethical concerns

All the data we use is publicly available for research. The work presented in our paper introduces no security, privacy, or ethical concerns.

A.4 Installation

Follow the following steps to set up the run-time enviroment required to run our code:

1. Clone the github repository and navigate into it:
   ```
   git clone https://github.com/pratik18v/
   robust-representation-matching.git &&
   cd robust-representation-matching
   ```
2. Create a Python virtual environment and activate it:
   ```
   conda create -n rrm python=3.6 &&
   conda activate rrm
   ```
3. Install dependencies:
   ```
   pip install -r requirements.txt
   ```
4. Install apex using instructions here:
   https://github.com/NVIDIA/apex#quick-start

All the instructions to setup the run-time enviroment are also provided in the GitHub README.

A.5 Evaluation and expected results

We demonstrate that our proposed algorithm (RRM) trains adversarially robust image classifiers faster than previous state-of-the-art method, at the same time attaining better robustness. For this, we train neural networks using several prior methods and compare them to our method. We perform comparison using two metrics: (1) total training time, and (2) accuracy on clean and adversarial test sets. We show that our method has the lowest total training time. Compared to the previous fastest method, our method trains classifier with higher adversarial accuracy. The accuracy numbers can be reproduced within a few percentage points of the numbers reported in the main paper. The absolute training time numbers will vary from what is reported in the main paper due to hardware differences. However, the speedup numbers (ratio of training times) can be reproduced to a value approximately similar to the reported value. The detailed steps to reproduce our results are laid out in the READMEs available in our GitHub repository.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

Our work leverages three artifacts to conduct our analyses. First, back-end data seized by Dutch National Police (DNP) from the Hansa marketplace. Second, data scraped externally by the research team. And third, code used to simulate artificial marketplaces. The back-end data is used to test the completeness and uncover biases in the scraped data. The simulation is used to explore how other scraping methodologies could achieve improved coverage, based on distributions from the Hansa back-end. To allow further research in online criminal marketplaces, we are making our public scrapes and simulation code available. However, all of our analyses of the back-end data were conducted on-site at Dutch law enforcement agencies, so we never stored nor owned the data ourselves. Due to Dutch privacy laws on law enforcement data we are thus unable to release that dataset.

A.2 Artifact check-list (meta-information)

- **Data set:** Yes, the data scraped externally is provided. The back-end data is not.
- **Publicly available (explicitly provide evolving version reference)?:** Yes, the scraped data can be visualized and queried at: https://arima.cylab.cmu.edu/markets/viewmarketplace.php?name=Hansa. The simulation code is found at: https://github.com/aledcuevas/dnm-simulation/releases/tag/v0.2
- **Security, privacy, and ethical concerns:** The scraped data contains no Personal Identifiable Information. Research using the scraped data should never seek to provide any legal proof of criminal conduct.
- **Code licenses (if publicly available)?:** The simulation has a MIT License.
- **Data licenses (if publicly available)?:** The scraped data has the following license: https://arima.cylab.cmu.edu/markets/license.php
- **Archived (explicitly provide DOI or stable reference)?:** The stable reference for the scraped data will be available at: https://www.impactcybertrust.org/dataset_view?idDataset=1498. Our DOI request is pending.

A.3 Description

A.3.1 How to access

- The simulation code can be cloned or downloaded from the following Github release: https://github.com/aledcuevas/dnm-simulation/releases/tag/v0.2.
- The anonymized version of our dataset can be queried and visualized by navigating to the following URL: https://arima.cylab.cmu.edu/markets/viewmarketplace.php?name=Hansa.

Downloads of anonymized and non-anonymized versions of our dataset are done through IMPACT Cyber Trust. A free account is required to download the dataset. Researchers pursuing legitimate R&D in a valid organization at a DHS-approved location are eligible for accounts. Accounts may take a few business days to be approved. For more details refer to: https://www.impactcybertrust.org/help_faq. Additionally, requests for the non-anonymized versions of the dataset are handled on a case-to-case basis and require the signature of a Memorandum of Agreement.

- The anonymized version of our dataset can be requested from the following URL: https://www.impactcybertrust.org/dataset_view?idDataset=1498.
- The non-anonymized version of our dataset can be requested from the following URL: https://www.impactcybertrust.org/dataset_view?idDataset=1499.

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

No software dependencies are needed beyond those shipped with the Python 3.8 standard library, `pandas`, and `numpy`. To use the Jolly Seber abundance estimator, `RMark` is required. Instructions on abundance estimators is available in the repository.

A.3.4 Data sets

No other data sets are required beyond those described in the "How to Access" subsection.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

The scraped data contains no Personal Identifiable Information. Research using the scraped data should never seek to provide any legal proof of criminal conduct.

A.4 Installation

No installation is required, assuming software dependencies are met. The code can be cloned or downloaded as a `.tar.gz` or `.zip`. The simulation code is executed by calling `main.py`. The datasets don't require any installation.

A.5 Evaluation and expected results

To evaluate the artifact, users should run `main.py`. We provide a set of test files that parameterize the simulation for testing purposes. The user will observe a count of days elapsed in `stdout`. The code will also create a folder structure (described in the GitHub documentation) which will contain the results of the simulation. Upon reaching the end of the simulation, the simulation will save to disk a `.json` file with a market

transcript (e.g., a record of the day that items, vendors, and reviews were created/hidden/deleted from the market). Additionally, the market will also run a simple artificial scraper which will output a `.json` file with the pages it captured as it scraped the market. Given that we are using dummy parameters, the expected result is a market transcript which contains between 1-10,000 vendor pages, 10,000-200,000 item pages, and 50,000 to 300,000 review pages.

A.6 Notes

While we provide a stable reference to our dataset, the DOI is still pending. Furthermore, given the number of days it may require to obtain an account from IMPACT Cyber Trust, we are happy to provide reviewers access to our raw anonymized dataset for evaluation through another channel, if necessary.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

This artifact contains the source code of RapidPatch and the stuff for running it. Since RapidPatch is designed for hotpatching embedded devices, to evaluate the basic functions, you need to have a Cortex-M3/M4 based arm development board. If you do not have these devices, we also provide a simple version that can run on qemu, and can demonstrate the functionable of RapidPatch by running the hotpatching process using fixed patch points (only one of the three hotpatching strategies supported by our tool). To fully evaluate and reproduce the results, you need to have at least one of these STM32F407/STM32L475/STM32F429/NRF52840 developing boards. Note that you can use any of the MacOS/Windows/Linux Platform to develop or evaluate it, we provide Docker and PlatformIO-based VSCode cross-platform building environments.

A.2 Artifact check-list (meta-information)

- **Binary: Pre-build RapidPatch firmware for different devices (you can also build from scratch).**
- **Hardware: Qemu and real devices, such as, STM32F429/NRF52840/STM32L475 and ESP32 developing boards.**
- **How much time is needed to prepare workflow (approximately)?: 3h**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: GPL v3.0**
- **Archived (provide DOI or stable reference)?: Yes**

A.3 Description

A.3.1 How to access

All the documents and source code are available on github.
https://github.com/IoTAccessControl/RapidPatch/tree/ae-v1.0.
(Commit: 591f82e5cf4f91cfa440bb376cb4975ce78ce871)

A.3.2 Hardware dependencies

RapidPatch relies on the Debug Monitor Handler of Cortex-M3+ MCU to dynamically trigger patches without modifying the Flash ROM. The recommended devices are NRF52840, STM32F429, or STM32L475. You can also port RapidPatch to other devices with Cortex-M3/M4 MCUs via PlatformIO.
Note that for devices other than Cortex-M3+, you can only use compiling time patch points placement.

A.3.3 Software dependencies

To compile the source code from scratch, you need to install the following software.

- Docker (manually)
- gcc-arm-none-eabi (installed by Docker)
- qemu-system-arm (installed by Docker)
- VSCode and the PlatformIO plugin (manually)
- Keil (optional)

If you do not have any required hardware and just want to quickly try it, we provide Docker scripts with a push-button to run the core functionalities of RapidPatch on any platform that supports Docker. In this case, you do not need to install any aforementioned software.

A.4 Installation

To run on Docker, you can use our docker images or build from the Dockerfile. The detail steps is shown in docker-qemu.md document.

To try RapidPatch on real devices, you can build and flash these projects with the Keil project or Platform-IO projects or just use the pre-build firmware.

A.5 Experiment workflow

You can follow the HOWTO.md document to test the functions of RapidPatch. There detailed steps of deploying a patch is as follows.

1. Integrate the RapidPatch Runtime to the firmware of your devices.

2. Write a patch based on the origin C source code patch.

3. Generate the eBPF bytecode via the RapidPatch Toolchain's patch generator.

```
python3 main.py gen test_cve1.c \
    test_cve1.bin
```

4. Verify the eBPF bytecode via the RapidPatch Toolchain's patch verifier. Note that, for the filter patch, the verifier can automatically insert the SFI instructions for loops.

```
python3 main.py verify test_cve1.bin
```

5. Deploy the patch to real devices with our Usart tool or directly paste the patches' bytecode to your firmware code.

```
python3 main.py monitor COM15
> install test_cve1.json
```

6. Test the patch functions with the Usart commend line interface.

A.6 Evaluation and expected results

After setting up the firmware, you can use a serial port tool (e.g., CoolTerm) to connect to the devices and trigger commends to conduct the evaluation. To preform the micro-evaluation, you need to use the Usart shell commend (e.g., run exp_idx) to execute the corresponding experiments.

The results of micro-benchmark is output to the Usart shell message and contains the execution time and CPU cycles.

```
Event 0 -> cycle: 38 time(us): 0.475000
```

To evaluate the macro-benchmark, you can use the the pre-built Zephyr Apps and the test tools to measure the performances. The results are written to local files.



A Artifact Appendix

A.1 Abstract

Our artifacts include the source code of all components of our Kage implementation, including the LLVM-based [36] compiler, the FreeRTOS-based [6] embedded OS, the microbenchmarks, the macrobenchmark, the binary code scanner, the corresponding libraries, and our scripts to find stitchable gadgets. Our hardware requirements include a host Linux machine and an STM32L475 Discovery board [43]. Our software requirements include Linux, a C/C++ compiler (e.g., Clang, gcc) and associated tools for compiling Clang and LLVM, the OpenSTM32 System Workbench IDE, Python 3, and the pyelftools library. We provide automated evaluation scripts to generate the performance results, code size results, and most of the security evaluation results included in the paper. Specifically, the performance results produced by these artifacts correspond to the results found in Tables 2, 3, 4, 5, 6, and 7 of the paper. Due to minor bug fixes and code structure adjustments, the artifact results will vary slightly from the results presented in the paper, but the key results and the main claims of the paper remain valid.

A.2 Artifact check-list (meta-information)

- **Program: CoreMark [28] (included), Microbenchmarks (included).**
- **Compilation: Our LLVM-based compiler.**
- **Transformations: Our compiler passes (shadow stack, store hardening, and CFI).**
- **Run-time environment: Fedora 35.**
- **Hardware: STM32L475 Discovery board.**
- **Metrics: CoreMark: Iter/s; microbenchmark: cycles; code size: bytes; security: gadgets.**
- **Output: Serial output containing the numerical results.**
- **Experiments: Execute the automated evaluation scripts.**
- **How much disk space required (approximately)?: 5GB.**
- **How much time is needed to prepare workflow (approximately)?: Two hours.**
- **How much time is needed to complete experiments (approximately)?: 20 minutes.**
- **Publicly available?: Yes.**
- **Code licenses: Kage, LLVM compiler, CoreMark: Apache License 2.0; Newlib: GNU General Public License 2; AWS FreeRTOS: MIT License.**
- **Archived?: https://github.com/URSec/Kage commit #195d489**

A.3 Description

A.3.1 How to access

The source code of Kage is publicly available as a GitHub repository: https://github.com/URSec/Kage.

A.3.2 Hardware dependencies

An STM32L475 Discovery board is required. Other STM32 development boards may work but are untested. A Linux x86 host machine is also required in order to build and flash the benchmarks to the board and to read the experimental results.

A.3.3 Software dependencies

We require the host machine to run a Linux distribution. We evaluated Kage using a host machine running the rolling release of Arch Linux, updated in June 2021. We have also tested Kage on Fedora 35.

Our build script uses the manufacturer-provided IDE to build the binaries. Therefore, we require the OpenSTM32 System Workbench IDE to be installed on the host machine. The IDE is publicly available at https://www.openstm32.org/HomePage. Note that users are required to register for a free web account to download the IDE suite.

Our binary code scanner requires Python 3 and the `pyelftools` library.

Finally, our automated evaluation script requires Python 3, the `colorama` Python library, and the `pyserial` Python library.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

A.4.1 Setting Up Kage on a Local Machine

We provide a detailed guide to install the dependencies and to set up Kage in the `readme.md` document of our GitHub repository.[5] As discussed in Section A.3.2, we require an STM32L475 Discovery board to run the compiled ARMv7-M binaries.

A.5 Experiment workflow

We provide a detailed guide to run the experiments in the `readme.md` document of our GitHub repository.

[5] https://github.com/URSec/Kage

A.6 Evaluation and expected results

A.6.1 Key Results in the Paper

There are three main claims in our paper. First, Kage incurs only minor performance overhead in the macrobenchmark, CoreMark [28], even though some of its components show a more significant overhead in microbenchmarks. Second, Kage incurs acceptable code size overhead. Third, Kage eliminates stitchable code-reuse gadgets.

For the first claim, the key result is that Kage incurs 5.2% mean performance overhead compared to the baseline FreeRTOS [6] in CoreMark. Table 3 in the paper lists the detailed CoreMark results. For the performance overhead of Kage's components, Table 5 and Table 6 in the paper list the microbenchmark results.

For the second claim, the key result is that Kage incurs 49.8% code size overhead compared to the baseline FreeRTOS and 14.2% code size overhead compared to FreeRTOS with MPU enabled, when comparing the CoreMark binaries that use three threads. Table 4 in the paper lists the detailed code size results.

For the third claim, the key result is that, for the CoreMark binaries that use three threads, Kage significantly reduces the number of reachable code-reuse gadgets and eliminates stitchable gadgets. Table 7 in the paper lists the detailed code-reuse gadget results for the security evaluation.

A.6.2 Reproducing the Results

As Section A.5 states, we provide a detailed guide to run the automated scripts in the `readme.md` document of our repository. This document includes detailed steps to build our toolchain, generate the performance and code size results, and generate the security evaluation results.

Because we discovered and fixed additional minor bugs in our workflow after we submitted the paper, and because we adjusted the source code to enable automated evaluation, the artifact results will include minor differences from the original results included in the paper. For the microbenchmarks, the results may include variations up to 25 cycles. For the performance evaluation of CoreMark, the results may include variation up to 0.05 Iter/s. For the code size evaluation of CoreMark, the code size of the untrusted code includes a difference of 16 bytes. Finally, for the security evaluation, the number of reachable gadgets includes a difference of one gadget. These differences do not significantly impact the key results and claims of the paper.

We note that our automated evaluation scripts produce a larger set of performance metrics than the set we included in the paper. For example, Table 6 in the paper shows the microbenchmark results for FreeRTOS, FreeRTOS with MPU enabled, and Kage. Our evaluation script, `run-benchmarks.py`, also shows the microbenchmark results for Kage's OS mechanisms. Similarly, for code size, the paper only includes the results of the CoreMark binaries that use three threads while the script also shows the code size results for the microbenchmark binaries as well as other binaries of CoreMark that use one or two threads.

Finally, while our scripts generate most of the results automatically, our security evaluation script, `run-gadget.py`, cannot automatically generate the number of stitchable gadgets because the process requires manual inspection. Section 6.2 of our paper explains how we analyze the reachable gadgets to determine if they are stitchable.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



B Artifact Appendix

B.1 Abstract

Our artifact contains source files of the Orca blocklisting protocol as a library in Rust. The cryptographic protocol is built on top of the open-source `arkworks` library for pairing-based cryptography. The implementation consists of three major parts: (1) an implementation of the Chase et al. algebraic MAC protocol, (2) an implementation of the Orca group signature, and (3) an implementation of the Orca one-time-use token protocol. The artifact also includes two benchmarks for reproducing the performance numbers reported on. These benchmarks can be easily run on any machine that can compile Rust from source, though we report performance numbers from running on high-memory AWS machines (for the server) and mobile devices (for the client). The artifact does not include source files for the griefing attack and battery-drain experiments against Signal, as they are potentially harmful and are not core to our work's claimed contribution.

B.2 Artifact check-list (meta-information)

- **Algorithm:** The Orca blocklisting protocol including group signature and one-time-use tokens.
- **Compilation:** Benchmarks are built from source using the Rust compiler.
- **Run-time environment:** Our artifact was run on a `c5.12xlarge` AWS EC2 virtual machine with 24 cores and 96 GB of memory running Ubuntu Server 20.04 LTS, as well as on a mobile device running Android 9.
- **Hardware:** The mobile microbenchmarks were run on a Google Pixel 2 device. The server throughput benchmark requires at least 64 GB, though comparable results can be reproduced with less memory.
- **Execution:** The microbenchmarks run in less than 5 minutes. The server throughput benchmark runs in under 2 hours on our test AWS machine.
- **Security, privacy, and ethical concerns:** We do not provide the source files for the griefing attack and battery-draining experiments.
- **Output:** The benchmarks produce summarized performance outputs printed to the terminal.
- **Experiments:** There are two benchmarks: (1) microbenchmarks for measuring the performance of the cryptographic primitives used in Orca, and (2) macrobenchmark for measuring server throughput of requests.
- **How much time is needed to prepare workflow (approximately)?:** The benchmark binaries are built from source in under 5 minutes. Setting up the AWS machine and/or the mobile device may take additional time.
- **Publicly available?:** The latest version of the library is available at `https://github.com/nirvantyagi/orca`. The version that underwent artifact review is marked with tag `usenix-sec22-ae`.

B.3 Description

B.3.1 How To Access

The latest version of the library is available at `https://github.com/nirvantyagi/orca`. The version that underwent artifact review is marked with tag `usenix-sec22-ae`.

B.3.2 Hardware Dependencies

Our artifact was run on a `c5.12xlarge` AWS EC2 virtual machine with 24 cores and 96 GB of memory running Ubuntu Server 20.04 LTS. The server throughput benchmark requires at least 64 GB, though comparable results can be reproduced with less memory. The mobile microbenchmarks were run on a Google Pixel 2 device running Android 9.

B.3.3 Software Dependencies

Full instructions for building from source are provided on the project README. All dependencies are readily available through the Rust package manager and binaries can be built from source in under 5 minutes.

B.3.4 Security, Privacy, and Ethical Concerns

We do not provide the source files for the griefing attack and battery-draining experiments.

B.4 Installation

The setup consists of installing Rust and compiling the benchmark binaries from source. Compiling and running the microbenchmarks on a mobile device requires additional installation of the Android Native Development Kit (NDK) and related Rust toolchains. The macrobenchmark for server throughput additionally requires installing and running a Redis server locally. Detailed installation instructions are given on the README available at `https://github.com/nirvantyagi/orca`.

B.5 Evaluation and Expected Results

There are two benchmark binaries that we report results on. The first is the microbenchmarks binary that is used to populate Figure 5. The platform and desktop client user columns are given from running the microbenchmark binary on a single core of the specified AWS machine. The mobile client user column is given from running the microbenchmark on the specified mobile device.

The second benchmark binary measures server throughput and is used to populate Figure 6. The reported numbers are based on experiments setting benchmark parameters of 200 requests for a blocklist size of 100, a strikelist size of 1400, and one million users, while varying the number of cores. This setup requires 64 GB of memory, however, the number of users can be reduced (e.g., to 200) to reproduce similar results without large memory requirements.

Detailed evaluation instructions are given on the README available at https://github.com/nirvantyagi/orca.

B.6 Experiment Customization

The benchmark source code is available and can be customized beyond the exisiting parameterization.

B.7 Notes

The cryptographic code has not been reviewed; it serves as a research prototype and is not suitable for deployment. If any bugs are discovered, please raise an issue on Github or send an email to the authors.



A Artifact Appendix

A.1 Abstract

This artifact contains the code for paper "Adversarial Detection Avoidance Attacks: Evaluating the robustness of perceptual hashing-based client-side scanning". We provide instructions for reproducing the results and running the attack on any other dataset of images. To reproduce the results in a timely fashion, we recommend using a Linux machine with at least 30 cores and 64G of RAM. A smaller machine will work but will require more time to run the experiments. The code is written to be modular such that it can be extended to run the attack on any other dataset. Furthermore, the experiments can be modified easily by changing a few parameters in the provided configuration files. This allows, for instance, to run the attack on a small number of images.

A.2 Artifact check-list (meta-information)

- **Algorithm:** The code for the attack algorithm is provided.
- **Data set:** ImageNet, the downloading and setting up instructions are provided.
- **Run-time environment:** Python
- **Metrics:** All metrics are provided in the code.
- **Output:** NPZ and PKL files containing the intermediate results. The final outputs are plots in PDF format.
- **Experiments:** All experiments are part of the code, and configuration files can be tuned to run them on a small scale.
- **How much disk space required (approximately)?:** 1TB.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours, includes the setting up of the Python environment and the downloading of the dataset.
- **How much time is needed to complete experiments (approximately)?:** 2 weeks.
- **Publicly available?:** No.

A.3 Description

A.3.1 How to access

The artifact will only be provided by the authors to the reviewers. We will not release it publicly for ethical concerns and sensitivity of the topic.

A.3.2 Hardware dependencies

We recommend using a Linux machine with at least 30 cores and 64G of RAM to be able to run the attacks and reproduce the results. A smaller non-Windows machine will also work but with significantly longer time required (e.g. 2 weeks) to run all the large-scale experiments included in the paper. Small-scale experiments (i.e., attacking a few images) can be run in a shorter time.

A.3.3 Software dependencies

We use Miniconda [1] to setup the Python environment. The libraries and instructions to setup the environment are provided in the README. The code is tested on linux machines (specifically Ubuntu 16.04 and Ubuntu 20.04). But we expect the code should run on any other OS without any change as no OS dependent code is used to the best of our knowledge.

A.3.4 Datasets

We use ImageNet for all the experiments. For the updated results with duplicates removed from the ImageNet dataset, we refer the reader to our extended arXiv version[2].

A.4 Installation

Installation is only required to setup the conda environment for Python. We provide all the instructions in the README along with environment yaml file.

A.5 Experiment workflow

The workflow involves

1. Setup the Python environment;
2. Downloading the ImageNet dataset;
3. Running the Python code (preferably in something like tmux) to compute the image hashes;
4. Running the experiments to generate pickle files with results;
5. Running a Jupyter notebook to generate plots from the results.

A.6 Evaluation and expected results

To reproduce the results we need to run the experiments with the provided configuration files. This in turn would generate pickle files with the results of each experiment. Then, the notebooks are used to generate plots from the paper and they also print the reported values. One can expect to reproduce all the results from the paper. For reproducibility We also provide the seed used for all the experiments. All the results and plots from our paper can be generated using this evaluation.

A.7 Experiment customization

The configuration files can be modified to reconfigure the experiments, e.g. running the attack on fewer images. Similarly, the modularity in the code enables us to extend the experiments to other hashing algorithms and datasets by simply inheriting the generic classes provided for each module.

[1] https://docs.conda.io/en/latest/miniconda.html
[2] https://arxiv.org/pdf/2106.09820.pdf



A Artifact Appendix

A.1 Abstract

E2SE is a system for securely storing private data in the cloud with the help of a key server (an App server). Our E2SE artifact is a prototype in Java including both the client and key server implementation. The software requirements includes JDK 8[1] or later, Maven 3.8.1 or later [2] and some dependencies which could be automatically downloaded by maven, and OpenSSL 1.1.1[3] with *libssl-dev*. To reproduce the evaluation results, the hardware requirements include an AWS EC2 t3.xlarge instance in Seoul for running the client, an AWS EC2 t2.micro instance in Osaka for running the key server, and a AWS S3 cloud server in Tokyo. We provide the EC2 instances satisfying the software requirements and S3 cloud server access for the evaluation.

The key server could be run to provide assistance for secure storage. Given a plain file, the client could run to securely deposit the file to cloud storage and securely retrieve it later. The client will output the time cost of each procedure. The average statistic result should be consistent with the efficiency part of our paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** OPRF, AES, KDF, SHA256
- **Program:** Siege[4], an open-source benchmarking tool used to test the performance of web server, is needed. The throughput test of key server in our paper is done with Siege 4.0.4.
- **Compilation:** JDK 8 or later, Maven 3.8.1 or laer
- **Run-time environment:** ubuntu 18.04 TLS.
- **Hardware:** An AWS EC2 t3.xlarge instance, an AWS EC2 t2.micro instance, and AWS S3 server are needed.
- **Run-time state:** It is network sensitive as the client needs to communicate with the key server and cloud server (AWS S3 in our implementation). Both the network delay between the client and key server & cloud server will affect the time cost. The client transfers the file to/from the cloud server, where the network speed also affect the measured time cost.
- **Execution:** The running time depends on the the file size and network delay and speed. In our experiment described in the paper, the time cost of running the whole procedure 25 times for each file varies from several minutes to one hour with the file increasing from 10mb to 300mb.
- **Metrics:** Running the compiled jar package with calling the key server, it provides service in port 20202 to help the client. Running the compiled jar package with calling the client, the execution time for each procedure is reported. Using Siege to test the key server performance, the throughput is reported.
- **Output:**
  1. For the efficiency test, the outputs are the running time of IBOPRF, Give, Take, optimized secure deposit and retrieve, secure deposit and retrieve, plain deposit and retrieve. The statistically average of the outputs should be consistent with the efficiency part of the paper, including the Figure 8,9 and the Table 2.
  2. For the key server throughput test. The output is the throughput of key server, say the number of transactions per second (trans/sec). When the key server is deployed on devices with different processing cores and memory, the throughput increases almost linearly with the processing core increasing.
- **Experiments:** The full preparation is described in the README.md instruction of the open source code https://github.com/yananli117/E2SE. We provide two well-prepared EC2 instances. The reviewer could upload the code to the instances, and follow the Run instruction and test instruction to get the results.
- **How much disk space required (approximately)?:** It depends on the data size. We test the files from 10mb to 300mb, so the required disk should be less than 1GB.
- **How much time is needed to prepare workflow (approximately)?:** Prepare from scratch, it probably costs 2 hours. We provide a well-configured instances in EC2 to run, only cost 10 minutes.
- **How much time is needed to complete experiments (approximately)?:** Several minutes are needed to test whether the artifact works. If redo all the experiments to produce the data of the four figures and one table about efficiency and scalability, approximately less than 6 hours are needed.
- **Publicly available:** . Github: https://github.com/yananli117/E2SE.
- **Code licenses:** Our code is under MIT license.
- **Archived (stable URL):** https://github.com/yananli117/E2SE/tree/bd4de7fb1c6c70df96bf89a17c100624fa665d0b

A.3 Description

A.3.1 How to access

We have open sourced our artifact at https://github.com/yananli117/E2SE. To reproduce the performance evaluation results, we provide the EC2 cloud instances with proper configurations and credentials, etc. Since we do not know when the reviewers will execute our code to repeat, to avoid keeping the cloud instances running for a whole month (which is a bit unnecessary waste), please inform us in the system before you plan to test. We will start the well-configured EC2 instances and send you the corresponding IP addresses.

A.3.2 Hardware dependencies

- To test the artifact is workable, two processes deployed in one or two devices are needed for the key server and client.

[1] https://www.oracle.com/java/technologies/javase/javase8-archive-downloads.html
[2] https://maven.apache.org/download.cgi
[3] https://www.openssl.org/source/
[4] https://github.com/JoeDog/siege

- To reproduce the evaluation results, we provides two EC2 instances for running the client and key server. The client is deployed in EC2 t3.xlarge instance in Soul, and the key server is deployed in EC2 t2.micro instance in Osaka.
- AWS S3 as cloud storage server (During the artifact review, we provide the access credential to access it). To apply to other cloud services, the code should be tuned a bit for the different cloud APIS.

A.3.3 Software dependencies

The software dependencies include JDK 8 or later version, Maven 3.8.1 or later versions, OpenSSL 1.1.1 and *libssl-dev*. (Some dependencies could be automatically installed in the compilation using Maven.) To test the throughput, the key server is implemented as a web server, so Tomcat + nginx framework are needed for the key server. The Siege tool in the test server is needed.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

When you plan to test the artifact, please do the following steps:

- Inform us to open the two instances and we will return the two ip addresses of the two instances for your access.
- Download all the credentials of AWS S3 with the link we provide only for artifact evaluation. For other users, please login in your own AWS account and get the security credential following the link https://docs.aws.amazon.com/general/latest/gr/aws-sec-cred-types.html.
- Open the terminal, securely and remotely control the EC2 t3.xlarge instance in Seoul via SSH for running the client

```
ssh -i CredentialPath/EC2_Client_Seoul.cer
    ubuntu@ip_client
```

- Open another terminal, and securely and remotely control the EC2 t2.micro instance in Osaka via SSH for running the key server

```
ssh -i CredentialPath/EC2_keyServer_Seoul.cer
    ubuntu@ip_client
```

- Clone the git repository and change to the root directory for both the client and key server

```
git clone https://github.com/yananli117/E2SE.git
cd E2SE/E2se4j
```

- Follow the instruction in README.md shown in https://github.com/yananli117/E2SE to config, compile, run and test our artifact and use our prototype for protecting data.

A.5 Experiment workflow

A.6 Evaluation and expected results

In our paper, we have two main claims efficiency and scalability.

A.6.1 Claim on efficiency

We do experiments to demonstrate that our design is efficient. We mainly measure the time cost of each procedure during the secure storage, including the register, give, take, deposit and retrieve procedures. We also compare the time cost of plain deposit/retrieve with the time cost of secure and optimized secure deposit/retrieve to show that the overhead of secure deposit/retrieve is very small, which could be seen in Figure 8,9 and Table 2.

When running the client with a specified plain file, 25 users run the whole procedure sequentially as follows: register to the system, run the give protocol to share the data encryption key (ibOPRF + give), encrypt the specified file and deposit the ciphertext to S3 in an optimized way (secureDepOpt), run the take protocol to reconstruct the data encryption key (ibOPRF + take), retrieve the ciphertext and decrypt it in an optimized way (secureRetOpt), encrypt the specified file and deposit the ciphertext to S3 (secureDep = Enc + DCT), retrieve the ciphertext and decrypt it (secureRet = RCT + Dec), deposit plain file to S3 (plainDep), retrieve plain file from S3 (plainRet), encrypt a plain file (Enc) and Decrypt the encrypted file (Dec).

We need to keep the key server running and run the client 8 times by specifying plain files of different sizes from 10mb to 300mb shown in the paper. To produce a file with specific size, we add the generation code in testGuide/ComFile1.java. Please follow the RREADME.md instruction to generate the file with a specific size.

With the output in the client terminal, we can calculate the average time cost for each procedure and the breakdown to form the Figure 8,9 and Table 2.

Since the time costs mainly comes from communication between the client and two servers, they could vary depends on the network delay between the deployed client to the deployed key server and the specified S3 server. The network speed could also affect the time cost especially when the size of file is large. So we cannot give the range of time cost for different network environments. We just claim that the test results could be reproduced if the experiments are the same as ours shown in the paper.

A.6.2 Claim on scalability

. We claim that our key server is scalable. We observe that in secure storage, the key server overhead mainly comes from interacting with the client to run the IBOPRF, which could affect the scalability. To demonstrate our key server is scalable, we deploy the key server as a web server with nginx + tomcat framework. We use Siege as the throughput benchmarking tool to test how many IBOPRF requests the key server could handle at per second. The client use Siege to sends 400 parallel https requests on IBOPRF to the key server and iterates 250 times. The specific commands are shown in README.md. only providing the IBOPRF service.

A.7 Experiment customization

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

The artifact contains the source code and installation scripts for the secure logging systems QuickLog and QuickLog2 in the paper. We also provided scripts to evaluate their application-independent signing and verification speeds, so that reviewers can reproduce the experiment results in Section 7.1 of the paper. We also included the code and scripts for installing and evaluating the competitor KennyLoggings.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** CentOS 7 (Linux version 3.10.0-1160.49.1.el7). We also tested our code on Ubuntu 18 (Linux 5.4.0-120-generic) to ensure that our code works with other Linux distributions. The code requires root access.
- **Hardware:** Our code requires that the machine supports AES-NI, which is generally available in modern CPUs.
- **Execution:** Our code runs in Linux. For the evaluation of the signing cost, we provide two separate sets of scripts for Linux version 5 and prior versions.
- **Metrics:** The evaluation scripts report the stand-alone execution time for the signing and verification operations.
- **Output:** For each iteration, the script runs the operation for 200,000 times and computes the median execution time. It runs for 10 such iterations, and outputs the median and standard deviation of those 10 median timings. Users can customize the message size.
- **Experiments:** We provide instructions for how to install our logging schemes in the Linux kernel and evaluate their signing and verification speeds in the `README` file of the github link below. This allows one to reproduce the experiment results in Section 7.1 of the paper.
- **How much disk space required (approximately)?:** 10MB.
- **How much time is needed to prepare workflow (approximately)?:** Two hours (for downloading the Linux kernel source code and patching the kernel).
- **How much time is needed to complete experiments (approximately)?:** 10 minutes.
- **Publicly available (explicitly provide evolving version reference)?:** Our code and scripts are publicly available at `https://github.com/TsongW/QuickLog/tree/1d1cb65ace83308306c1ae80e884a1f4ed68facd`
- **Code licenses (if publicly available)?:** GNU v3.0

A.3 Description

A.3.1 How to access

The code and scripts are publicly available at the github link above.

A.3.2 Hardware dependencies

Our code requires that the machine supports AES-NI, which is generally available in modern CPUs.

A.3.3 Software dependencies

Our code requires the availability of the source code of Linux kernel.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A.

A.4 Installation

- Download the Linux kernel source v3.10.0-1160.49.1.el7.
- Use the patches in the `patches` directory of the github link. Follow the guidelines to patch the Linux kernel at `https://wiki.centos.org/HowTos/Custom_Kernel`.

A.5 Experiment workflow

We provided scripts for compiling and benchmarking the schemes in the `README` file of the github link above.

A.6 Evaluation and expected results

The paper uses three benchmarks to evaluate the secure logging schemes; the artifact however only contains scripts to reproduce the first one. This benchmark measures the application-independent execution time of the signing and verification operations. For signing cost, we expect that (1) QuickLog and KennyLoggings have comparable performance for realistic log sizes (64B–384B), and (2) QuickLog2 is about twice faster than the other two schemes. For verification cost, we expect that (1) QuickLog and QuickLog2 have the same performance, whereas (2) KennyLoggings is 6–10 times slower. In our experiments, the standard deviation is within 5% of the median timing.

A.7 Experiment customization

N/A

A.8 Notes

The submission version of our paper contained only QuickLog. In the final version, we added an improved scheme QuickLog2 that has much faster signing time, better security, and no storage cost.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact contains a functional version of DepImpact and necessary data for the evaluation. The execution needs a virtual machine. The host machine may at least have 16GB memory and 64GB hard disk space. To facilitate the usage of this artifact, we prepare a linux virtual machine with necessary component to execute the artifact and visualize the result. Artifact users cam compare the result with our paper draft.

A.2 Artifact check-list (meta-information)

- **Algorithm:** No
- **Program:** Yes
- **Compilation:** No
- **Transformations:** No
- **Binary:** No
- **Model:** No
- **Data set:** Yes, contained in virtual machine
- **Run-time environment:** Ubuntu
- **Hardware:** No
- **Execution:** No
- **Metrics:** Please refer our paper draft
- **Output:** The graph and other necessary data
- **Experiments:** Artifact contains data for experiments
- **How much disk space required (approximately)?:** 30 GB
- **How much time is needed to prepare workflow (approximately)?:** 2 - 3 hours
- **How much time is needed to complete experiments (approximately)?:** 6 - 8 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** None
- **Data licenses (if publicly available)?:** None
- **Workflow framework used?:** None
- **Archived (provide DOI)?:** 10.5281/zenodo.5559214

A.3 Description

A.3.1 How to access

https://zenodo.org/record/5559214.YWYJT2LMKUk

A.3.2 Hardware dependencies

To effectively run the artifact, the host machine may at least need 16GB memory and 64GB hard disk spaces.

A.3.3 Software dependencies

No specific software dependencies for this artifact

A.3.4 Data sets

Virtual machine contains evaluation data. The DARPA TC raw data can be downloaded from its website.

A.4 Installation

Download Image file and import by the virtual machine.

A.5 Experiment workflow

In the virtual machine, there is a folder named **DepImpact-artifact** in the **home** directory, which contains two jar packages and a zip file.

- **DepImpact.jar** is used to generate the dependency graph from the log file and to filter out un-relative part for the POI event.
- **CalculateMissing.jar** is used to calculate false positive/negative rate based on the defined critical edges for each attack.
- **allcases.zip** contains logs and property files which are needed for the DepImpact as the input.

A.5.1 Command

java DepImpact.jar pathToRes pathToLog host logname1 logname2 ...

- **pathToRes:** the folder path of the output of DepImpact
- **pathToLog:** the folder path of the input of DepImpact
- **host:** **true** or **false** depends on the case that DepImpact needs to work with

A.6 Concrete Steps

1. Unzip allcase.zip folder
2. Create a folder for the output of DepImact (i.e. pathToRes)
3. Run listed commands:

- java -jar /home/artifact/DepImpact-artifact/DepImpact.jar pathToRes pathToLog false fileName.txt

  If the pathToLog is the path of the unzipped file from the first5cases.zip, the res folder case1 is for the attack Wget executable, the res folder case2 is for the attack Illegal Storage, the case3 is attack Illegal Storage2, the case4 is Hide File, the case5 is Steal information. If the pathToLog is the path of the unzipped file from the case67.zip, the res folder case6 is for the attack Backdoor Download, case7 is for the attack Annoying Server User. Information.
- java -jar /home/artifact/DepImpact-artifact/DepImpact.jar pathToRes pathToLog true logName.txt This command is for attack shellshock, Dataleak, and VPN Filter mentioned in our paper draft.
- java -jar /home/artifact/DepImpact-artifact/DepImpact.jar pathToRes pathToLog false fileName.dot This command is for the attack done by DARPA (Five Dir, Theia, and Trace).

Some cases may require huge memory, it may be suitable to run these cases on a powerful server. For the quick verification and try, we suggest reviewers run DepImpact on some small cases like Five Dir case1 or case3. Reviewers can take Table4 in our submission as a reference for the scale of different cases.

A.7 Evaluation and expected results

- The statistical information of dependency graph like node number and edge number will be in a file whose name ends with “json_log". In this log, it contains the number of node and edge after backtrack POI(Point of Interest), EdgeMerge, and time cost for each component of DepImpact.
- DepImpact will do forward analysis from top-ranked nodes. The filter result is under a folder whose name is DepImpact. The parent folder is set by **pathToRes**.
- The dependency graph is saved as a dot file. To show it, you may use the command:dot -Tsvg dotFilePath > svgFilePath

To calculate the false negative/positive rate for the attack, we need to provide identified critical edges for each attack. The critical edges for the attack used in the evaluation are defined in the corresponding property file.Users need to execute **calculateMissing.jar** to calculate the false positive/negative rate of DepImpact when using different numbers of top-ranked entry nodes. The command should follow this format: java -jar calculateMissing.jar path_of_the_DepImpact_outputs path_of_th*logtopN*

A.7.1 Concrete Examples

Example1:

1. run java -jar /home/artifact/DepImpact-artifact/DepImpact.jar /home/artifact/outputs /home/artifact/DepImpact-artifact/allcases/dataleak1 true dataleakhost1.txt

After this step, there will be a folder dataleak1-case1 created in folder /home/artifact/outputs.

The first thing we should do is to rename the folder "DepImpact" in folder dataleak1-case1 to "sysrep".

For the property file’s name we can ignore the part “-backward.property_", the folder name in the path_to_res(e.g. /home/aftifact/outputs) should be equal to the part befor "-backward.property_" plus "-" plus the part after "-backward.property_".

For this case, because the folder name dataleak1-case1 is not the same as the property file name dataleakhost1-backward.property_case1, we should change the folder name dataleak1-case1 to dataleakhost1-case1.

After this modification, run command:

java -jar /home/artifact/DepImpact-artifact/calculateMissing-1.0-SNAPSHOT-jar-with-dependencies.jar /home/artifact/outputs/dataleakhost1-case1 /home/artifact/DepImpact-artifact/allcases/dataleak1 2

then you will see some output in the terminal, at the same time, there will be a new json file created in the folder “sysrep".

Example2

run command: java -jar /home/artifact/DepImpact-artifact/DepImpact.jar /home/artifact/outputs /home/artifact/DepImpact-artifact/allcases/shellshock1 true shellshockhost1.txt

Modify the folder "DepImpact" in the folder shellshock1-case1 as "sysrep"

According to the property file, we need to modify the folder shellshock1-case1 to shellshockhost1-case1.

run command: java -jar /home/artifact/DepImpact-artifact/calculateMissing-1.0-SNAPSHOT-jar-with-dependencies.jar /home/artifact/outputs/shellshockhost1-case1 /home/artifact/DepImpact-artifact/allcases/shellshock1 2

then you will see some output in the terminal, at the same time, there will be a new json file created in the folder "sysrep".

A.7.2 Results explaintation

In file (eg “case-backward_json_log.json"), the key “BackTrackVertexNumber&EdgeNumbe" shows the number listed as Causality Anylysis #V & #E in Table 4 of our paper draft. The key “CPRVertexNumber& EdgeNumver " shows the number listed as Edge Merge #V & #E in Table 4 of our paper draft.



A Artifact Appendix

A.1 Abstract

Our data set contains nearly 10K binaries which are compiled by our toolchains to obtain the ground truth of binary disassembly. The binaries vary from x86/x64, arm32/aarch64 to mipsle32/mipsle64. To compare popular disassemblers with different ground truths, we also include the result of binary disassembly of popular disassemblers and ground truths in the data set. To validate the result of the paper, we prepare scripts to compare disassemblers with ground truth on major disassembly tasks(instruction recovery, function detection, and jump table reconstruction) and present the accuracy(precision and recall) in the console.

The minimal disk space is about 100 GB. We have tested it in `Ubuntu18.04` and `Ubuntu20.04`. The software requirements are `python3` and `python3-pip`.

A.2 Artifact check-list (meta-information)

- **Data set:** The data set contains 10K binaries and ground truths of binary disassembly. We open sourced the data set in https://doi.org/10.5281/zenodo.6566082. The approximate size is 85GB.
- **Run-time environment:** Linux. We tested in Ubuntu 18.04 and Ubuntu 20.04.
- **Metrics:** The accuracy(precision and recall) or the number of false positives and false negatives of disassemblers.
- **Output:** The output is shown in console. The result is numerical results. The expected result is shown in the paper.
- **Experiments:** We prepared `bash` scripts to automate the experiments as possible.
- **How much disk space required (approximately)?:** 100GB.
- **How much time is needed to prepare workflow (approximately)?:** 1-2 hour(s).
- **How much time is needed to complete experiments (approximately)?:** 9 hours.
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval
- **Code licenses (if publicly available)?:** MIT license.
- **Data licenses (if publicly available)?:** Creative Commons Attribution 4.0 International.
- **Archived (explicitly provide DOI or stable reference)?:** https://doi.org/10.5281/zenodo.6566082

A.3 Description

A.3.1 How to access

https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval

A.3.2 Hardware dependencies

As we prepared large scale data set and result of binary disassembly and ground truths, our artifact requires at least 100GB storage.

A.3.3 Software dependencies

`python3`, `python3-pip`, `docker`, `qemu`

A.3.4 Data sets

https://doi.org/10.5281/zenodo.6566082

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

- Download source code. https://github.com/junxzm1990/x86-sok. After downloading the source code, please change current directory to `x86-sok/artifact_eval`.
- Download data sets. (i) x86/x64 data sets are in https://zenodo.org/record/6566082/files/x86_dataset.tar.xz?download=1. The decompressed size is 56GB. (ii) arm32/aarch64 and mipsle32/mipsle64 data set is in https://zenodo.org/record/6566082/files/arm_mips_dataset.tar.gz?download=1. The decompressed size is 35GB. To evaluate the result easily, please create a new directory named `table_7` and move the second data set into it.
- Set up environment. Please refer to https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#set-up-environment.

A.5 Evaluation and expected results

- **Impacts on Training Accuracy.** To show the impacts on different ground truths for training accuracy, we evaluate instruction recovery of `XDA`. We prepared trained models of `XDA` and test suite. The expected result is shown in paper Table 3. The steps to reproduce the evaluation are in https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#xda1h.
- **Impacts on Tool Evaluation.** We evaluated `dyninst`, `ZAFL`, and `IDA` with different ground truths on x86/x64 testsuite. (i) The steps to reproduce the comparisons between `dyninst` with different ground truths of `dyninst` is in https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#performance-of-dyninst-on-complex-constructs40mins and the expected result is in paper Table 4. (ii) The steps to reproduce the comparisions between `ZAFL` with different ground truths is in https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#performance-

of-zafl-on-instruction-recovery1h and the expected result is in paper Table 5. (iii) The steps to reproduce the result of jump table recovery between `IDA pro` with `OracleGT` is in https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#distribution-of-precision-of-ida20mins and the expected result is in paper Figure 2.

- **Impacts on Tool Comparison.** We compared popular disassemblers on instruction recovery on `openssl`. The steps of reproduction is in https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#accuracy-of-popular-disassemblers-on-recovering-instructions20mins and the expected result is in paper Figure 3.
- **Impacts on improvements of `OracleGT`.** To show the impacts on improvements of `OracleGT`, we present the accuracy of popular disassemblers on recovering jump tables from `glibc`. The expected result is shown in paper Figure 5. The steps of reproduction are in https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#accuracy-of-popular-disassemblers-on-recovering-jump-tables-from-glibc10mins.
- **Evaluation of mainstream disassemblers on binaries with different architectures.** We present Figure 6 to show the recall and precision of mainstream disassemblers on binaries with different architectures. Note that the overall result in x86/x64 is nearly the same as the result presented in Sok [1], we skip the reproduction on binaries in x86/x64. To reproduce the result of arm32/aarch64 and mipsle32/mipsle64, we prepared the tutorial in https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#compare-result-of-arm-and-mips-disassemblers-result-3h. The expected result is shown in paper Table 7.
- `OracleGT` ***v.s.*** **Compilation Metadata** To reproduce the result, the tutorial is in https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#oraclegt-vs-compilation-metadata-20mins and the expected result is in paper Table 6.
- **Extendibility(Optional).** We also provide an example to show how to build a new test suite with our toolchains. The tutorial is in https://github.com/junxzm1990/x86-sok/tree/25656adbe14/artifact_eval#how-to-build-new-testsuite.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

References

[1] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851. IEEE, 2021.



C Artifact Appendix

C.1 Abstract

This artifact contains a functional version of PolyCruise and the necessary dataset for the evaluation. To facilitate the usage of this artifact, we have prepared a Docker image with the necessary components to execute the artifact and visualize the result. Artifact users can compare the results obtained from executing this artifact with those presented in our paper. It is recommended that the host machine has at least 16GB memory and 32GB hard disk space.

C.2 Artifact check-list (meta-information)

- **Algorithm:** No
- **Program:** Yes
- **Compilation:** No
- **Transformations:** No
- **Binary:** No
- **Model:** No
- **Data set:** Contained in the package
- **Run-time environment:** Ubuntu 18.04
- **Hardware:** No
- **Run-time state:** No
- **Execution:** No
- **Security, privacy, and ethical concerns:** None
- **Metrics:** Please refer to our paper
- **Output:** Textual information on the terminal
- **Experiments:** Necessary scripts provided
- **How much disk space required (approximately)?:** 10 GB
- **How much time is needed to prepare workflow (approximately)?:** 10 min
- **How much time is needed to complete experiments (approximately)?:** 20 min
- **Publicly available (explicitly provide evolving version reference)?:** Yes, https://github.com/Daybreak2019/PolyCruise
- **Code licenses (if publicly available)?:** Yes
- **Data licenses (if publicly available)?:** Yes
- **Archived (explicitly provide DOI or stable reference)?:** Yes, https://github.com/Daybreak2019/PolyCruise/releases/tag/v3.0

C.3 Description

C.3.1 How to access

- Download the Docker image for this artifact

  docker pull daybreak2019/polycruise:1.1

- Download the source code of PolyCruise

  git clone https://github.com/Daybreak2019/PolyCruise.git

C.3.2 Hardware dependencies

The host machine may at least need 16GB memory and 32GB hard disk spaces.

C.3.3 Software dependencies

PolyCruise is mainly developed on LLVM 7.0 and Python 3.7. Other software dependencies such as libxml and cmake are also necessary to build the project. For ease of use of PolyCruise, we have prepared a Docker image with all software dependencies installed.

Moreover, real-world benchmarks have their own particular/additional dependencies. Hence, to fully reproduce the results in the paper, users should install these dependencies successfully and ensure each benchmark's test cases can pass. For demonstration purposes, we use Cvxopt as a concrete example of such benchmarks and have installed all of its dependencies in the Docker image.

More specifically, we note that the Docker image includes all the libraries/framework underlying PolyCruise, thus it can be used for experimenting with other real-world subjects as well (i.e., saving the time/trouble for installing ubuntu, llvm, etc.) On the other hand, since our real-world subjects are sizable, including the complete compilation and run-time environment (e.g., all the third-party library dependencies) for all of them in the single Docker image would make it clumsy to deploy conveniently. Using a traditional virtual machine would aggregate this concern since they are even heavier. This is why we chose to include the setup for one such subject only inside the image at this time.

C.3.4 Data sets

PolyCruise is published with a set of micro-benchmarks included inside its code repository. The real-world benchmarks can be retrieved from GitHub.

C.3.5 Models

N/A.

C.3.6 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns with using this artifact.

C.4 Installation

- Step 1: Download the Docker image and run a Docker container based on the image

  docker pull daybreak2019/polycruise:1.1

  docker run -it daybreak2019/polycruise:1.1

Table 1: Effectiveness results of PolyCruise on PyCBench, including #inter-language paths (INT-LP), #Intra-language paths (ITR-LP), #false positives (FP), #false negatives (FN)

Group	#INT-LP	#ITR-LP	#FN	#FP
General flow	10	4	0	0
Global flow	9	0	0	0
Filed sensitivity	8	0	0	2
Object sensitivity	9	2	0	1
Dynamic invocation	4	0	0	0
Total	40	6	0	3

- Step 2: Clone the source of PolyCruise and build the project within the container

  cd /root/
  git clone https://github.com/Daybreak2019/PolyCruise.git
  cd PolyCruise
  ./build.sh

C.5 Experiment workflow

We provided scripts for automating all the experiments during the evaluation. Specifically in this artifact, we setup the environment for the experiments on the micro-benchmarks and a real-world Python-C program Cvxopt.

- Experiment on micro-benchmarks

  In this step, we can run all micro-benchmarks together with the following script:

  cd PolyCruise/PyCBench
  ./RunTest.sh

  If we want to run the micro-benchmarks one by one manually, we can switch to the appropriate directory and execute the script, for example:

  cd PolyCruise/PyCBench/GlobalFlow/3_leak_PyClang
  ./build.sh

- Experiment on Cvxopt

  cd PolyCruise/Experiments/scripts/cvxopt
  ./build.sh build

  The parameter "build" indicates the script to instrument and install Cvxopt before running the tests.

C.6 Evaluation and expected results

C.6.1 Evaluation on micro-benchmarks

We use micro-benchmarks (referred to as PyCBench in our research paper) to evaluate the effectiveness of PolyCruise, and it achieved 93.5% precision on PyCBench. The results are summarized in Table 1. Out of all of the 46 cases in PyCBench, PolyCruise succeeded in 43 and failed in 3.

Reproduction.

After building PolyCruise following the steps in Section C.4, use the following command to run PolyCruise on PyCBench:
cd PolyCruise/PyCBench
./RunTest.sh

Expected results.

Obtain the output in the end:
"Finish test, (Correct/Total) = (43, 46)"

Failed cases:

PyCBench/FieldSensitivity/7_leak_PyClang
PyCBench/FieldSensitivity/8_int-overflow_PyClang
PyCBench/ObjectSensitivity/11_leak_Python

C.6.2 Evaluation on Cvxopt

To verify the capability of cross-language vulnerability detection, we also demonstrate evaluating PolyCruise on real-world Python-C programs. In this artifact, we take Cvxopt as an example to exemplify our claim, PolyCruise succeeded in detecting real-world bugs and got one CVE assigned (Table 6 of our paper).

Reproduction.

Use the following command to build and run tests on Cvxopt:
cd PolyCruise/Experiments/scripts/cvxopt
./build.sh build

Expected results.

Report cross-language information flow paths of the `incomplete comparison` vulnerability as shown in Figure 1.

```
@@@@@@@@@@@@@@@@@@@@[69][deincmp]Reach sink,  EventId = 5 -- <Function:spsolve,  Inst:53>
                     [P (7F1DEF61E6B2,0)] [G (7F1DEF61DC2C,0)]
                      ---->case: deincmp spsolve 53
@@@@@ Ready to exit, total memory: 9092 (K)!
entry CheckCases ... (null)
@@@@@ GenSsPath -> Souece[1], Sink[1].....
[1 ][deincmp] Path: [F (Compute,0)] [A (F,0)]
                     [PY]pyinspect ->
                     [PY]Compute ->
                     [C]spsolve: [F (strncmp,0)] [U (Val2,0)] [P (7F1DEF61E6B2,0)] [G (7F1DEF
```

Figure 1: Example of result on Cvxopt

C.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

We have packed all the required software environments for reproducing our experiment results in a single docker image. The readers will rebuild the docker image and run the provided scripts to collect the results on time consumption, memory consumption, and code coverage. The collected results should match the records presented in the paper. To reproduce the results, the readers should have a server with at least 16GB RAM and 512GB storage space. Due to hardware discrepancies, the readers may observe some minor variations.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Linux, Docker
- **Hardware:** At least 16GB main memory and at least 512GB free disk space.
- **Metrics:** Execution time, memory consumption, and code coverage
- **Output:** Execution time and memory consumption records and raw generated inputs.
- **How much disk space required (approximately)?:** 512GB
- **How much time is needed to prepare workflow (approximately)?:** About 4 hours
- **How much time is needed to complete experiments (approximately)?:** About 7-14 days. The actual time needed depends on the hardware configurations.
- **Publicly available (explicitly provide evolving version reference)?:** Yes. We have made our code and scripts to reproduce the results in the paper publicly available at Github. The stable link pointing to the Git commit is https://github.com/R-Fuzz/fastgen/commit/01d31bc6bb42ee3535bb3aa8a0f88d345e9cb23d
- **Code licenses (if publicly available)?:** 3-clause BSD license
- **Data licenses (if publicly available)?:** 3-clause BSD license
- **Archived (explicitly provide DOI or stable reference)?:** Yes. Our code and scripts are archived in the Github. The stable link pointing to the Git Commit is https://github.com/R-Fuzz/fastgen/commit/01d31bc6bb42ee3535bb3aa8a0f88d345e9cb23d

A.3 Description

A.3.1 How to access

We have open-sourced all our code at https://github.com/R-Fuzz/fastgen/commit/01d31bc6bb42ee3535bb3aa8a0f88d345e9cb23d. We have also included the instructions to build the experiments environment and reproduce our results in the same repository.

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

Docker

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

We have provided a docker image which contains the complete environment for reproducing the experiments results.
To build the docker image, just run:

```
$ docker build -t usenix .
```

The details are under the "installation" section of the *README* file in the provided repository.

A.5 Experiment workflow

We have described the instructions for reproducing the results in the *README* file in the provided repository. A reader can follow the instructions, run the scripts and collect all the experiment results.

A.6 Evaluation and expected results

The main claim of the paper is that our system SymSan can significantly speed up the concolic execution and reduce the memory consumption. To support the claims, we conducted the experiments described in sections 5.2.1, 5.2.3, and 5.3 with the results presented in Table 2, Figure 4, Figure 5, and Figure 6 respectively.

By following the instructions in the *README*, a reader should be able to collect the results in Table 2, Figures 4,5, and 6. Specifically,

- To reproduce the results in paper's section 5.2.1, follow instructions in section 3.1.1 (nbench) of the *README*. The results in this section show that our system SymSan has a smaller instrumentation overhead than SymCC and SymQEMU.
- To reproduce the results in paper's section 5.2.3, follow instructions in section 3.1.2 (CGC) and section 3.1.3 (Real-world applications) of the *README*. The results in this section show that our system SymSan has a smaller performance overhead than SymCC and SymQEMU.

- To reproduce the results in paper's section 5.3, follow the instructions in section 3.2 (Memory consumption without solving) of the *README*. The results in this section show that SymSan has a smaller memory overhead than SymCC.

Because we perform all our evaluations on a server with an Intel(R) Xeon(R) E5-2683 v4 @ 2.10GHz (40MB cache) and 512GB of RAM, running Ubuntu 16.04 with Linux 4.4.0 64-bit, we recommend our readers hire a server with similar configurations for minimal variations.

The variations of performance numbers depend on the following factors: 1. The CPU frequency and size of CPU caches 2. The memory size and bandwidth, and 3. Whether or not the external storage is a hard disk or solid-state driver (for program loading). The maximum variations in absolute numbers should not exceed 100%. The variations for the relative numbers (SymSan's speed-up over SymCC) should not exceed 50%.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact

A.1 Abstract

In our Artifact, we provide the source code of CellIFT, a native RISC-V toolchain, and other dependencies. We also provide the framework for performing all the experiments described in this paper and analyzing the obtained results. Everything is packaged as a Docker image to allow for optimal reproducibility. To reproduce the experiments, we expect a machine with 256 GB memory and 500 GB of free storage.

A.2 Artifact checklist

- **Algorithm:** CellIFT is a newly developed algorithm to efficiently generate IFT shadow logic as part of a Yosys pass.
- **Program:** We use a set of five external RISC-V CPU designs (Ariane, BOOM, Ibex, Rocket, PULPissimo) as evaluation targets, as well as benchmarks from the RISC-V Architectural testing framework. All of this code is included in our artifact.
- **Compilation:** We include the required compilers and interpreters.
- **Transformations:** We include the required Verilog transformations (CellIFT and GLIFT), implemented as Yosys passes.
- **Binary:** We include prebuilt Verilator binaries of the five CPU designs in all instrumentation modes (i.e., vanilla, CellIFT, and GLIFT) where possible. Note that GLIFT instrumentation or synthesis sometimes fails, as explained in Section 7.2.
- **Run-time environment:** The bulk of our artifact is a Docker image that runs on Linux. We tested our image on an Ubuntu 22.04 system with 5.15.0-37-generic kernel.
- **Hardware:** We do not require any special hardware, but do need a relatively large amount of DRAM (256 GB) to run all the experiments.
- **Metrics:** The experiments record runtime performance and IFT precision for microbenchmarks for CellIFT as well as GLIFT. Further experiments record execution time and memory footprint of the instrumentation and synthesis process for all instrumentation modes. We also measure the simulation performance on for all instrumentation modes. Lastly, we show resource usage and clockable frequency after FPGA synthesis for all the five CPU designs under all instrumentation modes.
- **Output:** For all experiments used in the Evaluation section of this paper (Section 7), we include code to regenerate the charts. Also, we include code to reproduce all results in the Scenarios section of this paper (Section 8).
- **Experiments:** With the exception of the FPGA results, all experiments are executed automatically when building the Docker image. This means the way to reproduce all experiments is encoded in the Dockerfile, and a Docker container based on this Dockerfile would contain the generated results, and can be used to re-run individual experiments if desired.
- **How much disk space required:** The docker image with all the layers is 330 GB, and Xilinx Vivado requires around 150 GB for downloading and installation. In total, we estimate a total of 500 GB of free storage is required.
- **How much time is needed to prepare workflow:** To prepare the workflow, conscious effort is only needed to retrieve the Git repository and the Docker image, which should take only a few minutes.
- **How much time is needed to complete experiments:** Reproducing the experiments takes approximately 3 days.
- **Publicly available:** Stable URL: https://github.com/comsec-group/cellift-artifacts/commit/eaa9a26ae85fd6a7ae8cd248416315414ae4c135. The README points to a stable (sha256-verified) Dockerhub Docker image that contains the rest of the code and data, namely docker.io/ethcomsec/cellift-artifact-evaluation@sha256:9a15d4070d321026ad4d5d9ba5a236842c6c456279f9c08f4fa4132de7b399ce.
- **Code licenses:** CellIFT is licensed under GPL3.
- **Workflow frameworks used:** Docker, Make, Luigi.

A.3 Description

A.3.1 How to access

The project is located at https://comsec.ethz.ch/cellift. Our artifact is a single Git repository designed primarily to build a Docker image that has run all the experiments automatically. This Git repository is hosted at the 'Publicly available' checklist entry. The README.md in that repository contains further instructions to obtain the prebuilt Docker image from Dockerhub.

A.3.2 Hardware dependencies

The artifact will run all experiments on a machine with 256 GB of memory.

A.3.3 Software dependencies

We tested the Docker image on Ubuntu 22.04 LTS kernel 5.15.0-37-generic, but we expect it to work on a wide range of Linux distributions.

To reproduce the FPGA experiments in the paper, we furthermore depend on the Xilinx Vivado FPGA synthesis tool (version 2019.3).

A.4 Installation

The installation of our artifact requires the following two steps:

1. Cloning the git repository specified in the checklist and using its README.md to pull the Docker image artifact hosted on Dockerhub.
2. Reproducing the FPGA experiments, requires the installation of the full edition of Vivado 2019.3 from the Xilinx website and a license.

A.5 Experiment workflow

Follow the instructions in the git repository README.md that specifies in detail how to start a Docker container with the image, and how to reproduce each experiment, and examine the results.

In principle, cloning the git artifact repository and rebuilding the Docker image using the Dockerfile in the git repository will rebuild all CELLIFT code and designs from scratch and perform the experiments (except the FPGA experiments). For maximum reliability, we also provide the prebuilt Docker image with all code, binaries and results that we have found to work, which can be used to reproduce all the experiments (and use CELLIFT in general if desired).

To run the FPGA experiments, first source the settings64.sh file from the Vivado installation dir, and follow the instructions in the Artifact README.md.

A.6 Evaluation and expected results

The key results from our experiments are as follows. For each result, we point to scripts (Python or bash) that drive the experiments and show the analysis.

1. Instrumented designs that we can synthesize to C++ (i.e. be compiled) for all five RISC-V CPU designs, contrary to GLIFT, and with less CPU time and memory (follows from plot_instrumentation_performance.py and plot_rss.py), and with higher tainting precision (follows from plot_num_tainted_states_ibex.py).
2. For the designs that can be compiled in all instrumentation modes, we show that CELLIFT has lower performance overhead than GLIFT (follows from plot_benchmark_performance.py).
3. The Meltdown and Spectre simulations reproduce Figure 11, showing they can both be detected (follows from plot_tainted_elements.py).
4. We show several bug scenarios detected by CELLIFT (run_scenarios.sh).
5. We show FPGA synthesis results, showing that CELLIFT instrumented designs can be synthesized, with fewer resources than the GLIFT instrumented designs.

We refer to the README.md of the artifact git repository for the detailed steps to reproduce each of the key results described above.

A.7 Experiment customization

There is ample customization opportunity in the Docker image, because the code of the instrumentation tool as well as the target designs are there and can be modified and rebuilt. This does require a deeper knowledge that goes beyond this appendix.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

We implement a GPU-based information flow query tool and make its source code public for access. The project is available at https://github.com/mimicji/FlowMatrix/. In this Artifact Evaluation, we are applying for:

- "Artifact Available" badge

For "Artifact Available" badge application, we do not require hardware and software for this badge evaluation. However, to run our tool, NVIDIA GPU(s) and NVIDIA CUDA toolchain are required. In the paper, we claim the source code is available on GitHub (on page 6, Implementation Section). Thus, the "Artifact Available" badge would support our claims of availability.

A.2 Artifact check-list (meta-information)

- **Algorithm:** We do not propose any new algorithm.
- **Program:** We do not use public benchmark.
- **Compilation:** Besides the general compiler (GCC>=7.5), we also require a public compiler for our project: NVCC, the CUDA compiler driver. The version of the NVCC must be or greater than 11.3.
- **Transformations:** We do not require program transformation tool.
- **Binary:** We do not provide binaries.
- **Model:** We do not include models.
- **Data set:** We used recent CVEs and popular open-source programs for evaluation. The exploit of CVEs can be found in NVD.
- **Run-time environment:** We require Ubuntu (>=16.04). The `glibc` version should be greater than 2.30. We do not require root access.
- **Hardware:** We require special hardware for running our project: NVIDIA GPU(s). Any NVIDIA GPUs that CUDA toolkit supports would meet the requirement. However, we evaluate FlowMatrix with two V100 cards and different hardware other than V100 provided may affect the performance of FlowMatrix.
- **Run-time state:** Our tool is not sensitive to run-time state.
- **Execution:** We do not have specific conditions for execution. However, to gain the best performance, we recommend running FlowMatrix as the only task on GPUs.
- **Security, privacy, and ethical concerns:** There is no specific security, privacy, or ethical concerns.
- **Metrics:** The reported metrics include execution time, data flow throughput, performance profiling.
- **Output:** The tool will show the results to the console in numbers.
- **Experiments:** We have system scripts to run the experiments automatically.
- **How much disk space required (approximately)?:** Around 900GBs.
- **How much time is needed to prepare workflow (approximately)?:** Around 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** Around 20 hours, depending on the devices.
- **Publicly available (explicitly provide evolving version reference)?:** The source code for FlowMatrix can be found at https://github.com/mimicji/FlowMatrix. The submitted version is https://github.com/mimicji/FlowMatrix/tree/c4a809f6c76ac447d0baf542db9e04b8d4600436.
- **Code licenses (if publicly available)?:** The code license is GPL3.0. Check the file `LISENSE` for more details.
- **Data licenses (if publicly available)?:** Not provided.
- **Workflow frameworks used?:** No workflow frameworks are used.
- **Archived (explicitly provide DOI or stable reference)?:** Please check https://github.com/mimicji/FlowMatrix/tree/c4a809f6c76ac447d0baf542db9e04b8d4600436.

A.3 Description

A.3.1 How to access

Please follow this URL https://github.com/mimicji/FlowMatrix/tree/c4a809f6c76ac447d0baf542db9e04b8d4600436.

A.3.2 Hardware dependencies

Any NVIDIA GPUs which are supported by CUDA.

A.3.3 Software dependencies

NVIDIA CUDA toolkit, SQLite C++ package, Protocol Buffers C++ package, Capstone C++ version.

A.3.4 Data sets

Not provided.

A.3.5 Models

No models are used.

A.3.6 Security, privacy, and ethical concerns

No Security, privacy, and ethical concerns.

A.4 Installation

After download, just run `make` command at the project home directory.

A.5 Experiment workflow

After compilation, a folder named `bin` with executable binaries will be generated under the project home directory. A system script has been used to automatically initialize the tool, pre-proceed all traces in the database, query the information flows with specified sources and destinations, and finally report results and performance numbers to the console. Following are the steps if a user would like to run the experiment step by step manually (which is also the workflow of the auto script).

Step 1: Initialization. To run FlowMatrix, the user needs to specify an SQLite database file for storage as the only parameter in the command line:

$$\$\ ./bin/QueryCLI\ [path_to_database]$$

This will open FlowMatrix console. Next, the user can choose one trace project stored in the database to work with in the console:

$$FMQuery>\ WorkOn\ [TraceName]$$

The user may check the list of traces via *ListTraces* command:

$$FMQuery>\ ListTraces$$

Step 2: Pre-processing. If the trace has never been pre-proceeded (pre-summarized), users need to tell FlowMatrix to pre-proceed parts of or all of it:

$$FMQuery>\ PrepareQuery\ [start]\ [end]$$

To check the trace length, we suggest the commend *TraceSize* which shows users the current trace length:

$$FMQuery>\ TraceSize$$

Also, although traces have their own trace reader, FlowMatrix allows users to view traces in the console:

$$FMQuery>\ TracePrintRange\ [start]\ [end]$$

Step 3: Querying. Once Step 2 has been done, the trace is ready to be queried for its information flows. The users may specify a sub-range of the pre-proceeded range to query information flows using *Query* command:

$$FMQuery>\ Query\ [V1Type]\ [V1]\ [Direct]\ [V2Type]\ [V2]$$

In this command, we support four types of data variables (*V1Type, V2Type*) to be queried, including system calls, instructions, registers, memory slots. Detailed usage can be found in the README.md document at https://github.com/mimicji/FlowMatrix/blob/main/README.md#query-usage. In the experiments, we usually choose *mmap*, *read* and *receive* system calls as the sources for CVEs, depending on the exploit. We choose input system calls or simply random instructions as the sources of a common program. The destination differs from CVEs, which can be *write* and *send* system calls, a register, or a memory slot. When it comes to common programs, destinations are output system calls and random instructions.

A.6 Evaluation and expected results

The claims have been made in this work:

- We analyze offline dynamic information flow operations on binaries and identify their linearity property.
- We propose FlowMatrix, a novel way of representing DIFT operations using matrices that enabling off-the-shelf GPUs to be used as a hardware co-processor for DIFT.
- We design an efficient solution to support interactive DIFT queries on offline execution traces. Our prototype demonstrates sub-second response time for several DIFT queries in common DIFT workloads.
- Our tool is open-sourced at GitHub.

First two claims are elaborated in the manuscripts and do not require experiments to support them.

The third claim can be supported by the performance results. Once the auto script finishes, it reports the performance for pre-processing and queries. The expected results can be found in the paper. The variation of empirical results depends on the provided GPUs and hard disks. In our test environment with two NVIDIA Tesla V100 cards and SSD, the pre-processing may have a 20% variance while the variance of query time may be 50% (especially in small cases).

The fourth claim can be supported by link access.

A.7 Experiment customization

NA.

A.8 Notes

NA.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact appendix describes the complete workflow to setup Bedrock. It includes an artifact check-list, description of hardware/software dependencies, experiment workflow, and the expected results.

A.2 Artifact check-list (meta-information)

- **Compilation:** GCC v7.5.0, Tofino SDE v8.4.0, Netronome SDE v6.1.0.
- **Binary:** Source code included to generate binaries.
- **Run-time environment:** End host codes are tested on x86 servers running 64-bit Ubuntu18.04 OS with BPF compiler collection. Both servers and switch need root access.
- **Hardware:** Intel/Barefoot Tofino1 switch ×1, x86 server with Mellanox ConnectX-4 RNICs ×4, x86 server with Netronome Agilio CX ×1.
- **Metrics:** Throughput, latency, CPU utilization, attack and defense effectiveness.
- **Output:** The server, client and attacker programs output messages indicating whether the attack succeeds. Throughput and latency can be measure by tools like `tcpdump`. CPU utilization can be measurement via tools like `top`.
- **Experiments:** See Section A.5 and Section A.5.
- **How much disk space required (approximately)?:** 1GB (dependencies not included).
- **How much time is needed to prepare workflow (approximately)?:** Compiling all programs needs about 1 hour (installation of software dependencies and hardware is not included).
- **How much time is needed to complete experiments (approximately)?:** About 2 hours to see the effect of all attacks and defenses.
- **Publicly available?:** Yes, code is available on GitHub.
- **Code licenses:** MIT license

A.3 Description

A.3.1 How to access

Bedrock is publicly available at the following GitHub repository: https://github.com/alex1230608/Bedrock. (commit: 4eef2619d7fb007b4c8ed690c6d78e8fea377455)

A.3.2 Hardware dependencies

To run Bedrock, it requires four x86 servers connecting to an Intel/Barefoot Tofino switch or a Netronome Agilio CX SmartNICs through Mellanox ConnectX-4 RNICs.

A.3.3 Software dependencies

Our experiments are performed on x86 servers running 64-bit Ubuntu 18.04, but similar Linux distributions should also work. To enable RDMA, Mellanox MLNX_OFED driver must be installed on the servers. Bedrock's P4 code is compiled by proprietary toolchains provided by the switch and SmartNIC vendors.

A.4 Installation

We list the main steps to install Bedrock here. More details can be found in our GitHub repository.

- Install the BPF Compiler Collection (BCC) on end hosts for eBPF module compilation and loading.
- Install RNIC drivers to enable RDMA on end hosts.
- Install and setup the programmable switch and SmartNICs following the vendor instructions.

A.5 Experiment workflow

We briefly summarize the workflow of running experiments on Intel/Barefoot Tofino switches in Bedrock; detailed instructions can be found in the provided README in our GitHub repository. Note that all experiments in Bedrock share the similar workflow as described in the following.

1. **Compile P4 program:** Compile the P4 programs using Intel/Barefoot Tofino switch SDE.
2. **Run Bedrock or baseline:** Run P4 programs on the switch. Both Bedrock's programs (i.e., `switch/bedrock_*.p4`) and the baseline program (i.e., `switch/baseline.p4`) are provided.
3. **Load eBPF modules (for authentication experiments only):** Load Bedrock's eBPF module on the RDMA servers and clients for authentication experiments.
4. **Setup the logging server (for logging experiments only):** Setup and start the logging server for logging experiments.
5. **Compile and run RDMA servers, clients, and attackers:** All needed end host programs (i.e., RDMA server, RDMA client, and attacker) can be compiled with `make` in the corresponding folders. Folder and file names are summarized in Table 4.

A.6 Evaluation and expected results

We evaluate Bedrock in different attacking scenarios. The following describes the expected results:

Authentication: We uses attack S1 to demonstrate effectiveness of proposed source authentication in Bedrock. When the experiment starts, the server terminal will keep dumping the memory content. The attack can be launched by setting the victim client's QPN, PSN, and rkey in the attacker program. Without Bedrock (`baseline.p4`), the memory content will keep changing, indicating that the attacker illegally accesses the memory. By deploying Bedrock, the attack will be blocked and the memory content will remain the same.

ACL: Bedrock enables more flexible ACLs inside the network. In this experiment, when Bedrock is not started, all RDMA

Experiment	Folder	Server	Client	Attacker
Auth.	`authentication`	`server_auth`	`client_auth`	`client_attacker`
ACL	`authorization/attack_demo`	`server_acl`	`client_acl`	N/A
Mon.-BW	`monitoring/bw_exhaustion`	`victim`	`client`	`client`
Mon.-QP	`monitoring/qp_exhaustion`	`victim`	N/A	`attacker`
Log.	`logging/pythia_attack_demo`	`server`	`client`	`client`

Table 4: The folder and file names of end host programs for each experiment.

requests will get responses (printed on the client terminal). Bedrock enables operators to deploy new ACL rules and block RDMA requests violating the rules. If a request is blocked by Bedrock, the client will show `Completion status 12`.

Monitoring—bandwidth exhaustion: The effectiveness of the attack and defense is evaluated by the bandwidth usage of traffic from each client (refer to our Github repository for details). Results should be similar to Figure 6(b) in the main paper where the attack starts at t=2.7s and Bedrock mitigates the attack at t=5.4s.

Monitoring—QP exhaustion: In this experiment, attackers try to consume as many QPs (queue pairs) as possible to cause QP exhaustion. Without Bedrock, the attacker can keep creating queue pairs on the server until the server cannot allocate queue pairs anymore. With Bedrock, a single client can only consume a predefined number of queue pairs. When the attacker tries to ask for more, the IP address will be banned and no further RDMA traffic will be allowed from that user.

Logging: We demonstrate the logging system in Bedrock by detecting and mitigating the Pythia side-channel attack (see more details in the main paper). Running the experiment will output the accuracy of the attack both at the terminal and in the output folder `logging/pythia_attack_demo/output`. Without Bedrock, the accuracy can be as high as 95%, but Bedrock will detect and mitigate the attack.



A Artifact Appendix

A.1 Abstract

Our artifact consists of 1) the SBAS client and node code used to operate the SBAS infrastructure 2) the simulation software used in the security analysis section of the paper and 3) the raw survey results and questions from our network operator survey.

A.2 Artifact check-list (meta-information)

- **Program:** Our artifact contains two programs: 1) the SBAS node and client software that is used to operate the SBAS infrastructure and connect clients respectively and 2) the topology simulation software to run inter-domain topology simulations are perform the security analysis.
- **Compilation:** The SBAS client and node must be compiled and installed as per the instructions in the README.md file. The README.md file (in the usenix22 branch) also contains instructions for setting up a personal SBAS using two nodes that are connected over SCIONLab (and have connectivity to each other but are distinct from our current SBAS production deployment). Additionally, the README.md file contains instructions for how to connect a client this SBAS deployment. The topology simulator is written in python and can be run directly on general-purpose computing hardware and requires no compilation (although the script to graph the results requires several pip and apt dependencies).
- **Binary:** The simulation code is in python (which is interpreted) so there is no binary, but the primary source code file is simulate.py in the root directory of the simulation repo. The SBAS client and node are largely python and bash scripts and the repo contains an install script that installs SBAS as a systemd service.
- **Data set:** Our SBAS client does not require any datasets. The simulation artifact uses the CAIDA AS relationships dataset, RIPE NCC and RouteViews BGP datasets, and PEERING testbed connection data. Our survey result dataset is attached as part of our artifact submission.
- **Run-time environment:** Our simulation requires python3 and the appropriate pip3 modules installed. While our code should run on most Linux environments, **all our testing was done on Ubuntu 22.04** and this was used to generate the required dependencies and install instructions mentioned in the README.md files. We strongly encourage Ubuntu 22.04 as other variants might require different package dependencies and even other Ubuntu versions ship with different versions of python that could potentially impact script behavior.
- **Hardware:** Simulations are run using general purpose hardware.
- **Execution:** The simulation models interdomain routing attacks and outputs statistics about the security of SBAS nodes which can be graphed as a CDF (see the README.md file in the usenix22-simulations branch). The SBAS node and client software install systemd services that manage routing rules related to forwarding SBAS traffic and interact with the other routing services SBAS depends on (e.g., the SCION-IP gateway and BIRD).
- **Security, privacy, and ethical concerns:** Simulations are run on static data/configuration files and thus pose no burden on the ASes and prefixes modeled therein. They are also based entirely on publicly-available datasets. The SBAS node and client software does not violate any networking best practices and only sends IP packets for common well defined protocols.
- **Output:** The simulation outputs result files for standard and ROV SBAS experiments. The SBAS node and client software does not produce any output file per say but configures routing such that secure prefixes and customers can be reached between different SBAS nodes.
- **How much disk space required (approximately)?:** The SBAS client and node software requires only minimal disk space for dependencies to install. Running the abridged version of the simulation requires under 1GB of disk space.
- **How much time is needed to prepare workflow (approximately)?:** Simulation workflow requires only time needed to download the topology simulator repository.
- **How much time is needed to complete experiments (approximately)?:** The simulation workflow requires roughly 1 hour using a general purpose CPU.
- **Publicly available (explicitly provide evolving version reference)?:** The artifact contents are hosted at https://github.com/scion-backbone/sbas/tree/80044509e5ac1681e8d970a09e4b3187439a0938. The client and node software and configuration files are available in the `sbas` submodule; the simulation code and data, in the `sbas-simulation` submodule; and lastly, the survey results in `sbas-survey`.
- **Code licenses (if publicly available)?:** CC Zero.
- **Workflow frameworks used?:** Github.
- **Archived (explicitly provide DOI or stable reference)?:** A stable link to our artifact is available at https://github.com/scion-backbone/sbas/tree/80044509e5ac1681e8d970a09e4b3187439a0938

A.3 Description

Obligatory. For inapplicable subsections (e.g., the "How to access" subsection when not applying for the "Artifacts Available" badge), please specify 'N/A'.

A.3.1 How to access

Artifacts are available online at the URLs listed in A.2

A.3.2 Hardware dependencies

Standard computational hardware.

A.3.3 Software dependencies

Python 3.7 and standard numerical/data science packages (numpy, matplotlib, pandas). See README.md files for more details on dependency installs (which can all be done with standard packages).

A.3.4 Data sets

We rely on the CAIDA AS-relationship data set and BGP update data from RIPE NCC and RouteViews, to generate the policy files and topology used for the simulation.

A.3.5 Models

N.A.

A.3.6 Security, privacy, and ethical concerns

The simulations employ only publicly available datasets and thus do not leak any private information about interdomain connectivity.

A.4 Installation

Obligatory. Describe the setup procedures for your artifact targeting novice users (even if you use a VM image or access to a remote machine).

The full installation instructions for the node and client are included in the `sbas` submodule under the artifact repository at the URL given in A.2 (specifically, https://github.com/scion-backbone/sbas/tree/80044509e5ac1681e8d970a09e4b3187439a0938). The installation for the simulation software is described in the `sbas-simulation` submodule.

A.5 Experiment workflow

Describe the high-level view of your experimental workflow and how it is implemented, invoked and customized (if needed), i.e. some OS scripts, IPython/Jupyter notebook, portable CK workflow, etc. This subsection is optional as long as the experiment workflow can be easily embedded in the next subsection.

The BGP simulation framework takes three main files as input:

1. **CAIDA Topology:** serial-2 AS Relationships topology
2. **policies file:** BGP export/import policies to be applied to BGP announcement points
3. **origins file:** enumerates prefix announcements to be made at BGP announcement points (including hijackers)

The simulation engine runs as a Python script and writes the outcome of each simulation scenario to a text file. Another Python visualization script generates a CDF of the simulation results similar to those presented in Figures 8 & 9 of the main paper. The full experimental workflow for the simulations is described in the sbas-simulation submodule of https://github.com/scion-backbone/sbas/tree/80044509e5ac1681e8d970a09e4b3187439a0938.

For the SBAS node software, the workflow involves joining SCIONLab, connecting two machines to SCIONLab, running SBAS on those two machines and testing connectivity through SBAS and then connecting a client to one of the machines and testing connectivity to the other SBAS node. The full workflow is described in the sbas submodule of https://github.com/scion-backbone/sbas/tree/80044509e5ac1681e8d970a09e4b3187439a0938

A.6 Evaluation and expected results

Obligatory. Start by listing the main claims in your paper. Next, list your key results and detail how they each support the main claims. Finally, detail all the steps to reproduce each of the key results in your paper by running the artifacts. Describe the expected results and the maximum variation of empirical results (particularly important for performance numbers).

We package a subsample of the data used in the Internet-scale simulations (presented in Section 7.2) to model the BGP hijack resiliency gains of SBAS provides over a client making its own BGP announcements to the Internet. These simulations should output textual data showing the proportion of the Internet that will be affected by an adversary's attack for each SBAS announcement. This output is then parsed by the plotting script `plot_artifact_results.py` to plot CDFs illustrating the resilience of SBAS against different adversaries.

There are several main results which can be seen in this CDF plots even with the reduced input files used for the artifact evaluation. First, an SBAS announced-prefix has a significantly higher resilience than a non-SBAS prefix. Second, SBAS performance improves with additional nodes are added to the network of PoPs. Finally, ROV enforcement further improves the resilience of SBAS.

The primary expected result for the SBAS node code is that the pings from one SBAS node's VPN prefix to another SBAS node's VPN prefix (which are routed over SCIONLab) are sent successfully. Furthermore, for the client code, the client connected to the SBAS node should be able to ping the VPN prefix at the other SBAS node securely which implies it could communicate with other clients connected to that node's VPN.

A.7 Experiment customization

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact includes a binary release of our proposed disassembler, DeepDi, and a sample script to show how we can use it to obtain instruction and function boundaries. This binary release is mainly to demonstrate the performance and efficiency of DeepDi, so for now it can only be used on Windows where CUDA is available.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Windows 10 x64, CUDA 11.1, cuDNN v8.2.4, MSVC runtime 2019, and Python 3.8.
- **Hardware:** NVIDIA GPU with compute capability of 5.2 or above.
- **Metrics:** Execution time and validation accuracy.
- **Output:** The evaluation script outputs precision, recall, and execution time.
- **Experiments:** An evaluation script is provided.
- **How much disk space required (approximately)?:** 300 MB.
- **How much time is needed to prepare workflow (approximately)?:** An hour to download and install all the dependencies.
- **How much time is needed to complete experiments (approximately)?:** A minute.
- **Publicly available?:** Yes, on GitHub.

A.3 Description

A.3.1 How to access

The full artifact is available on GitHub at the following URL: https://github.com/DeepBitsTechnology/DeepDi/tree/74f0af0d4cdf33fc5de6f55d5f4ec5142de68c18.

A.3.2 Hardware dependencies

A machine with NVIDIA GPU with compute capability of 5.2 or above is required. Here is a list of NVIDIA GPUs and their compute capabilities: https://developer.nvidia.com/cuda-gpus.

A.3.3 Software dependencies

- Windows 10 x64
- Python 3.8
- CUDA 11.1
- cuDNN v8.2.4 for CUDA 11.4
- MSVC runtime 2019
  - https://aka.ms/vs/16/release/vc_redist.x64.exe
- pyelftools
  - pip install pyelftools
- NumPy

A.3.4 Data sets

This artifact contains the benchmark and real-world binaries and the corresponding ground truth for instruction- and function-level evaluation, but feel free to try any x86 or x64 binaries. Please change DATA_PATH in eval.py to your real data path.

You can download the benchmark binaries at https://drive.google.com/file/d/1UfS4YsbKWw6Xlp7NXf4tTHDN7gzDRY7p/view?usp=sharing and the real-world binaries at https://drive.google.com/file/d/1x3N_0FAmSuT56D-KHRPvSaMz8Xl3gQFj/view?usp=sharing.

A.4 Installation

This binary release requires no installation. Detailed instructions on using and running the tool are included in the README file.

A.5 Experiment workflow

The included evaluation script gets the code section of the included sample file and feeds it into DeepDi. DeepDi then outputs where instructions and functions are.

A.6 Evaluation and expected results

The evaluation script will measure the precision and recall of instruction and function prediction, and the execution time of DeepDi.

A.7 Experiment customization

Though only one binary is included for demonstration purposes, DeepDi can be easily extended to evaluate arbitrary x86 and x64 binaries.



A Artifact Appendix

A.1 Abstract

In this artifact we compare three performance degradation strategies on Intel CPUs. In particular we measure the performance impact of performing a cache-flush based performance degradation in Intel microarchitectures with HyperThreading support. This artifact can be used to reproduce Tables 8-9 in the paper *"HyperDegrade: From GHz to MHz Effective CPU Frequencies"*. It can be also employed to extend the comparison to other microarchitectures.

A.2 Artifact check-list (meta-information)

- **Benchmark:** BEEBS
- **Compilation:** GNU toolchain
- **Hardware:** Intel with HyperThreading
- **Metrics:** clock cycles
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 2–50 hours
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** 10.5281/zenodo.5549559

A.3 Description

A.3.1 How to access

We provide full documentation in `README.md` available at the following URL. https://doi.org/10.5281/zenodo.5549559

A.3.2 Hardware dependencies

1. Intel CPU
2. HyperThreading
3. Recommended: Skylake, Kaby Lake, Coffee Lake, or Whiskey Lake

A.3.3 Software dependencies

1. Linux (root)
2. GNU toolchain
3. git
4. perf
5. python3

A.4 Installation

See `README.md` at https://doi.org/10.5281/zenodo.5549559.

A.5 Evaluation and expected results

1. This artifact reproduces the results in Section 4 of the paper.
2. In particular, Tables 8-9 in the paper.



D Artifact Appendix

D.1 Abstract

The artifact consists of the full source code of Pacer's prototype and instructions for building from source. In addition, we provide applications, datasets, scripts, and instructions for reproducing two sets of results from the paper: Pacer's bandwidth overheads and its empirical security evaluation.

D.2 Artifact check-list (meta-information)

- **Algorithm:** video-clustering, doc-clustering, CNN classifier
- **Model:** custom CNN classifiers (included in the repository)
- **Data set:**
  (1) clustering dataset: csv containing sizes of files in the corpus,
  (2) attack dataset: network traffic traces to run classifier on
- **Run-time environment:** Linux, python
- **Execution:** manual
- **Metrics:**
  (1) bandwidth overheads vs privacy (cluster size)
  (2) attack performance (classifier accuracy, precision, recall)
- **Output:** table, graphs
- **Experiments:**
  (1) clustering (cluster size vs. bandwidth)
  (2) classifier prediction
  (3) Video latency
  (4) Medical service throughput and client latencies
- **How much disk space required (approximately)?:**
  All source code: 30GB
  Total including compiled models and dataset: 50GB
- **How much time is needed to complete experiments (approximately)?:** 24 hours in total
- **Publicly available (explicitly provide evolving version reference)?:** https://gitlab.mpi-sws.org/pacer/pacer
- **Code licenses (if publicly available)?:**
  Pacer: MIT
  Linux: GPLv2
  Xen: GPLv2 Apache HTTP Server: Apache License 2.0
  wrk2: Apache License 2.0
  Mediawiki: GPLv2
  Memcached: BSD license
- **Data licenses (if publicly available)?:**
  Wiki datasets: CC-BY-SA
- **Archived (explicitly provide DOI or stable reference)?:**
  https://gitlab.mpi-sws.org/pacer/pacer/-/tags/security22-ae

D.3 Description

D.3.1 How to access

The artifact is publicly available at: https://gitlab.mpi-sws.org/aasthakm/pacer

D.3.2 Hardware dependencies

To reproduce the runtime performance results, Pacer must be set up on servers with a Broadcom Corporation NetXtreme II BCM57800 1/10 Gigabit Ethernet NIC and with bnx2x driver.

D.3.3 Software dependencies

Pacer's prototype relies on:

- Xen: 4.10.0
- Linux: 4.9.5
- OS: Ubuntu 16.04 LTS
- gcc: 5.4.0

Experimental evaluation has been done with the following software:

- Apache HTTP Server: 2.4.33
- Mediawiki: 1.27.1
- Memcached: 1.6.9
- OpenSSL: 1.1.0g

D.3.4 Data sets

Relevant datasets are provided as part of this artifact.

D.3.5 Models

Models are provided as part of this artifact.

D.4 Installation

Instructions are provided at: https://gitlab.mpi-sws.org/pacer/pacer/-/blob/main/install.md

D.5 Experiment workflow

Experiments can be run using the scripts provided in the repository. All the instructions are provided at: https://gitlab.mpi-sws.org/pacer/pacer/-/blob/main/experiments.md

D.6 Evaluation and expected results

We provide prepared artifacts to reproduce results of Pacer's bandwidth overheads and empirical security evaluation:

- Bandwidth overhead (section 6.2): https://gitlab.mpi-sws.org/pacer/pacer/-/tree/main/eval/bandwidth
- Attack classifier performance evaluation (section 6.4, appendix A): https://gitlab.mpi-sws.org/pacer/pacer/-/tree/main/eval/attack

D.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

This artifact includes the gem5 simulator and McPAT files that are used for performance impact and area/power estimation for the paper "Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks". The aforementioned frameworks will generate result files which we explain how to extract the relevant results in the appendix. In order to provide the artifact with minimal dependencies, we have provided the reviewers with a safe remote access to a server that contains all of the relevant dependencies, benchmarks, executables, source code and scripts, all of which can be inspected, overwritten and executed.

A.2 Artifact check-list (meta-information)

All of the following material in the checklist is included in the server. There is nothing to be downloaded. We are going to be mentioning them for context.

- **Algorithm:** The main algorithms we introduce are the allocation and the remapping code in the gem5 simulator. The bulk of our cachelet allocation algorithm is contained at `gem5/src/mem/cache/tags/indexing_policies`. In this directory, we have a base class in `base.cc` where the allocation and class constructor reside. For the remapping algorithm, we modify `set_associative.cc` where in the enclave mode, we emulate way deflection by returning the ways allocated for the enclaves to the replacement policy (in function `getPossibleEntries`). For the indexes of the cache we use the function named `extractSet` where we replace the higher bits with the VPT entry.
- **Program:** In the server, we include the following benchmarks (with corresponding versions) for your access:
  - SPEC2017 - 1.0.2 - private
  - PARSEC - 3.0-beta-20150206 - public
  - MiBench - 1.0 - public
  - Post-Quantum Cryptography (PQC) programs: BIG QUAKE, CRYSTALS-KYBER, CFPKM, Compact-LWE, DAGS - N/A - public
- **Compilation:** During the evaluation process, you are not obligated to compile anything. If desired, we have included compilation steps for gem5 in the Section A.8 in the appendix. The artifact already possesses every dependency for the compilation which are the same as the baseline gem5 CPU simulator.
- **Transformations:** There are no transformation tools required.
- **Binary:** The main binary file for the simulations is already pre-generated (`gem5/build/X86/gem5.opt`) on the platform. The McPAT simulators used in our evaluation also have pre-compiled binaries in `area_power_estim/mcpat` and `area_power_estim/mcpat_extra_tag`. The purposes of the two versions of McPAT are described in Section A.6.2.
- **Run-time environment:** On the reviewer's side, any OS that has SSH can run remote access the environment we provide and run all simulations. For context, the environment on the server is Debian 8.
- **Hardware:** We use the gem5 CPU simulator, an open source architecture simulator which is already included in the server (we don't require any installation or any extra hardware).
- **Execution:** The only condition we have is the establishment of the SSH connection to our platform. Simulations take few hours to few days depending on the experiment.
- **Security, privacy, and ethical concerns:** There are no security implications on the reviewer-side.
- **Metrics:** For performance metrics, we consider normalized Instructions Per Cycle (IPC). As for area/power estimation, we consider mm^2 for area and Watts for power generated by McPAT, an integrated power, area, and timing modeling framework for various architectures.
- **Output:** The files and directories that contain the metrics are explained in detail in Section A.6.
- **Experiments:** The experimentation process is explained in detail in Section A.5 .
- **How much disk space required (approximately)?:** While there's no requirement on the disk-space on the reviewer's machine, the platform memory has to be kept track of.
- **How much time is needed to prepare workflow (approximately)?:** No time needed to prepare the workflow.
- **How much time is needed to complete experiments (approximately)?:** Depending on the experiments, it can take a few hours to 3 days where full system simulations (like PARSEC experiments) take days and system emulation of security benchmarks (like PQC and MiBench security programs). The McPAT estimation experiments take a few seconds to complete.
- **Publicly available (explicitly provide evolving version reference)?:** No.
- **Code licenses (if publicly available)?:** No.
- **Data licenses (if publicly available)?:** No.
- **Workflow frameworks used?:** We use custom scripts that spawn shell commands to be run concurrently. The McPAT simulations are run manually from the command line.
- **Archived (explicitly provide DOI or stable reference)?:** No.

A.3 Description

A.3.1 How to access

N/A

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

The only software required is Secure SHell (SSH) which we are going to use to connect to the artifact platform.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

A.4 Installation

We do not require any reviewer-side installation other than SSH (please refer to online sources for proper installation). The artifact is going to be accessed remotely. One can access our servers through:

```
ssh <username>@<dns_tag>
```

where the `username`, the DNS tag and the password are granted in the submission.

We use our platform as the artifact testing environment. First off, the reviewer is going to access the server (required information provided in artifact submission form).

Once accessed, we highly recommend that you create a `tmux` session so that in case of an unintended disconnect or any other disturbance, the experiment process can keep running independent from the remote user. To create a tmux session, the following command line can be used:

```
tmux new -s <name_of_session>
```

We have already created a session named `reviewer`. In order for a user to attach to a session, the following command line can be used:

```
tmux a -t <name_of_session>
```

To detach from a session, press "Ctrl+B" and then "Q" on your keyboard; this will send you back to the main terminal interface. When attached to a session, the user will run terminal command to run the experiments we set up. This way, if you run an experiment on the tmux session, it will keep running on the server even when you disconnect.

For PARSEC experiments, we use gem5's full system simulation for multi-threaded applications. We have already prepared the disk image and the kernel image for this simulation. The gem5 version we use requires the path to the simulator as an environment variable; so after each connection or session creation where a PARSEC-related simulation is going to be run, please set the environment variable `M5_PATH` as:

```
export M5_PATH=/home/reviewer/gem5
```

Keep in mind that you don't need to do this constantly if your are running on a prepared tmux session.

A.5 Experiment workflow

We have 2 different simulation environments. The first one is the performance results gathered from the gem5 CPU simulator which we modified to emulate Composable Cachelets.

A.5.1 Performance Results from the gem5 Simulator

Experiment Scripts Explained We have three Python scripts for each benchmark suite: `runspec.py`. `runmi.py`, and `runparsec.py`. These scripts create simulation threads for various configurations by calling the gem5 binary (`gem5.opt`). On top of that, for non-enclave simulations, we use `runnonenc.py`.

gem5 is modified to have extra convenience options to the baseline such as `l3_vpt_size` that defines the number of VPT entries. Another example is `l3_cachelet_assoc` which defines the associativity of allocated cachelets. The user will not be interacting with these options directly, but it is important to note.

SPEC2017 Enclave Experiments For SPEC2017 benchmark suite evaluations, we use `runspec.py`. It tests 14 benchmarks from the suite and tests one 10 cachelet configurations along with the baseline configuration. The script has 3 mandatory options which are `real_insts`, `warmup_insts`, and `jobs` where they denote the number of real instructions (instruction considered for performance evaluation), number of instructions considered as initialization (these are ignored) and number of concurrent experiments. The platform we provide has 48 cores and all of the experiments are required, we recommend that you initialize `-jobs` as 40. Having said that, to run 1 billion real instruction and after 1 billion warm up instructions with 40 concurrent jobs, the following command would be executed:

```
python3 runspec.py --real_insts=1000000000
    --warmup_insts=1000000000 --jobs=40
```

Unfortunately, we cannot support further customization. For spec benchmarks. Please refer to the source code of the scripts for further customization.

Security Enclave Experiments For security benchmarks (MiBench and PQC), we have `runmi.py` where it takes only the same `jobs` argument. This script runs 4 configurations (including the baseline) per 8 benchmarks we consider.

The following command is an example of security benchmark experiment with 40 concurrent jobs:

```
python3 runmi.py --jobs=40
```

PARSEC Enclave Experiments Finally, for PARSEC benchmarks, we have a similar option layout for the script (`runparsec.py` in this case) to the security benchmarks. However, since PARSEC experiments are done in full system simulation we recommend 3 concurrent jobs at maximum. There is only 1 configuration per benchmark, and we consider 5 of them. To run PARSEC benchmarks, the following command line can be run:

```
python3 runparsec.py --jobs=2
```

SPEC2017 Non-Enclave Experiments For non-enclave experiments, we considered 5 SPEC benchmarks. The script we use for these experiments has the same option layout as `runspec.py`. To run these experiments:

```
python3 runnonenc.py --real_insts=1000000000
    --warmup_insts=1000000000 --jobs=40
```

is going to be used as the command.

A.5.2 Area and Power Estimation from McPAT

The files involved in this experimental workflow are located in `/home/reviewer/area_power_estim/`, under the following subdirectories:

- `mcpat`: contains the unmodified mcpat simulator source code and binaries from the public git repository.
- `mcpat_extra_tag`: contains a version of the mcpat source and binaries that we modified to simulate additional cache tags. The source differs from the baseline `mcpat` only in the `/cacti/const.h` header file, where the value of the `EXTRA_TAG_BITS` variable was changed from 5 to 9.
- `cc_descriptions`: contains the architecture descriptions used to run the simulations.
- `cc_mcpat_final_results`: contains the area/power result files from which we extracted the estimates.

We estimated the area and power overheads for major CC components (the VPT, CFL, and extended cache tags) using the `mcpat` computer architecture simulator. We describe the configurations used for our simulations below. The results of the simulations, and the procedures for running them, are presented in Section A.6.2

Baseline processor: As a point of comparison for our area and power results, we took one of the default processor specification files (Intel Xeon processor) provided with mcpat, and modified the Caches, Register File, TLB, BTB, LSQ, ROB, and fetch/decode/issue/commit width parameters to match the architecture used elsewhere in our evaluations. This file is located at:

```
/home/reviewer/area_power_estim/cc_descriptions/
    cc_base_processor.xml
```

VPT and CFL: We observed that the components VPT and CFL components of CC are closely analogous to existing hardware components simulated by McPAT. The VPT is comparable to a register alias table (retirement RAT), while the CFL corresponds to a register free list. Thus, we used the McPAT area and power metrics for a Integer Retirement RAT and Free List as our estimates for the area and power of the VPT and CFL, respectively. When generating results for each component, we configured the simulation so that the components would have a similar size and organization to VPT and CFL described in our paper:

- For the configuration that generated the VPT results (`/home/reviewer/area_power_estim/cc_descriptions/cc_single_issue_vpt.xml`), we specified 16 architectural registers and 64 physical registers. The resulting RAT would be comparable to a 16-entry VPT in a CC system with 64 total cachelets. Because a VPT does not require multiple write ports, we also gave the processor a single instruction width, so that the rename logic has a single write port.
- To obtain a 64-entry register free list (comparble to the 64-entry CFL assumed in our area/power evaluation), we modified the baseline processor specification to have 64 physical registers. The configuration file used for the CFL results was `/home/reviewer/area_power_estim/cc_descriptions/cc_free_list.xml`

Extended Cache Tags: To estimate the area and power overhead of the additional cache tag bits required by CC, we changed the number of extra cache tag bits defined in the McPAT source code. We then recompile the simulator. Specifically, we edited the file

```
/home/reviewer/area_power_estim/mcpat_extra_tag/
    cacti/const.h
```

to change the value of the `EXTRA_TAG_BITS` variable from `5` to `9`, reflecting the number of extra tag bits needed to support up to 16 cachelets per enclave. Using the baseline processor configuration `cc_base_processor.xml`, we ran the simulation with the modified and the unmodified versions of McPAT, then took the difference between the total area and power results for each simulation to determine the area and power increase attributable to the the added tag bits.

A.6 Evaluation and expected results

A.6.1 Performance Results from the gem5 Simulator

After the simulation ends, gem5 generates output directories which contain the architecture parameters and results. In these directories there are two important files, namely `config.ini` (architectural parameters) and `stats.txt` (computational metrics). `stats.txt` contains the IPC values that we are going to inspect. The scripts we use generate such output directories with names with the prefix `m5out`. Hence, to extract the IPC values from any benchmark, we use grep commands such as:

```
grep "switch_cpus_1.commit.committed_per_cycle::mean"
    artifact_cc/m5out_*/stats.txt
```

To elaborate, this command will print out all of the IPC results. Ideally we would like to specify the configuration or the benchmark of the experiments we desire to examine. We explain below how to extract IPC values from all experiments.

SPEC2017 Enclave Outputs To extract IPC values from a specific benchmark (say `deepsjeng`), the following commands should be run (baseline case first, other cases for the latter):

```
grep "switch_cpus_1.commit.committed_per_cycle::mean"
    artifact_cc/m5out_deepsjeng_baseline/stats.txt
```

```
grep "switch_cpus_1.commit.committed_per_cycle::mean"
    artifact_cc/m5out_deepsjeng_enclave*/stats.txt
```

If we want to compare different benchmarks with the same configuration (say 4 way 8 cachelets), the following command should be run:

```
grep
"switch_cpus_1.commit.committed_per_cycle::mean"
    artifact_cc/m5out_*enclave_4_8/stats.txt
```

Security Enclave Outputs As an example, to extract IPC from the `aes` benchmark, the following command should be run:

```
grep
"switch_cpus_1.commit.committed_per_cycle::mean"
     artifact_cc/m5out_aes_encrypt*/stats.txt
```

MiBench related benchmarks' (blowfish, sha and aes) directories are followed by an `_encrypt` suffix. PQC related benchmarks do not have such suffixes. Thus, for instance, getting IPC values from the benchmark BIG QUAKE is done by:

```
grep
"switch_cpus_1.commit.committed_per_cycle::mean"
    artifact_cc/m5out_BIG_QUAKE*/stats.txt
```

PARSEC Enclave Outputs PARSEC benchmarks are saved with a `parsec_` prefix, so to extract results from the benchmark `blackscholes`:

```
grep
"switch_cpus_1.commit.committed_per_cycle::mean"
artifact_cc/m5out_parsec_blackscholes*/stats.txt
```

Non-Enclave Outputs Non-enclave experiments have `_nonenclave` suffix. So, for inspecting `omnetpp` with 12 ways the following command should be used:

```
grep
"switch_cpus_1.commit.committed_per_cycle::mean"
artifact_cc/m5out_omnetpp_nonenclave_12/stats.txt
```

So to check the same benchmark for all cases:

```
grep
"switch_cpus_1.commit.committed_per_cycle::mean"
artifact_cc/m5out_omnetpp_nonenclave*/stats.txt
```

Expected Results For expected results, please refer to the main paper where every detail is already discussed in detail. Since we are using simulations, there might be slight differences to the results in the paper. However, the patterns the reviewers extract shall generate the same patterns as the ones in the paper, even though the process is not fully-deterministic.

	area mm^2 (% base)	peak W (% base)	runtime W (% base)
Base arch	45.183 (100)	70.0737 (100)	35.1191 (100)
VPT	0.00042 (0.00093)	0.0022 (0.0031)	0.0066 (0.019)
VPT × 2	0.00084 (0.0019)	0.0044 (0.0063)	0.013 (0.037)
CFL	0.019 (0.042)	0.060 (0.085)	0.057 (0.16)
Tag bits	0.36 (0.80)	0.21 (0.29)	0.11 (0.33)

Table 1: Results of McPAT simulations of CC components. Peak and runtime refer to peak dynamic and runtime dynamic, Percentages are relative to the baseline architecture (Base arch)

A.6.2 Area and Power Estimation from McPAT

Our area and power estimates indicated that the main CC hardware components impose a modest overhead in terms of processor area and power. Table 1 presents our results. Note that the same results are in the submitted paper were rounded to a different level of precision.

These results above were obtained as follows:

Baseline Architecture The area and power metrics for the baseline architecture were obtained with the following command:

```
cd /home/reviewer/area_power_estim/mcpat
./mcpat -print_level 5 -infile ../cc_descriptions/
    cc_base_processor.xml > ../
    cc_mcpat_final_results/cc_5_tag
```

The "Base arch" results shown in Table 1 come from the "Processor" section of the output file.

VPT estimates The area and power metrics for the VPT were obtained with the following command:

```
cd /home/reviewer/area_power_estim/mcpat
./mcpat -print_level 5 -infile ../cc_descriptions/
    cc_single_issue_vpt.xml > ../
    cc_mcpat_final_results/cc_single_issue_vpt_res.
    txt
```

The "VPT" results shown in Table 1 come from the "Int Retire RAT" section of the output file. Note that in the paper, we multiply the results by two to reflect the presence of two VPTs in a two core processor.

CFL estimates The area and power metrics for the CFL were obtained with the following command:

```
cd /home/reviewer/area_power_estim/mcpat
./mcpat -print_level 5 -infile ../cc_descriptions/
    cc_free_list.xml > ../cc_mcpat_final_results/
    free_list_as_64_entry_cfl.xml
```

The "CFL" results shown in Table 1 come from the "Free List" section of the output file.

Tag bit overheads The tag bit overheads were obtained with the following command:

```
cd /home/reviewer/area_power_estim/mcpat_extra_tag
./mcpat -print_level 5 -infile ../cc_descriptions/
    cc_base_processor.xml > ../
    cc_mcpat_final_results/cc_9_tag
```

The "Tag bits" results shown in Table 1 are the difference between the values in the "Processor" section of the `cc_9_tag` output file and the corresponding values in the baseline `cc_5_tag` file.

A.7 Experiment customization

We provide customization in terms of number of concurrent jobs and instruction numbers (when necessary). Yet, Due to the sheer number of the considered benchmarks, we cannot provide extensive customization for benchmark specification. However, if some benchmarks are not needed by the reviewers, they can be commented out.

A.8 Notes

If desired, the reviewers can compile gem5 (which is not necessary). To construct gem5, after entering the gem5 subdirectory by:

```
cd gem5
```

the following command should be used:

```
python2 `which scons` build/X86/gem5.opt -j 40
```

Unless any change is applied to gem5, this command will print the message:

```
scons: 'build/X86/gem5.opt' is up to date.
```

Also, please make sure that only one reviewer runs a specific experiment at a time to prevent interference, since aforementioned tools/frameworks overwrite namesake files/directories.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact contains our reverse-engineering (RE) tools, the covert and side-channel attacks, and the analytical model described in the paper. The RE tools can be used to explore the mesh interconnect by testing various sender/receiver placements. After RE, the covert and side-channel proof-of-concepts can be run. All of our tools were tested on a bare-metal machine with Ubuntu 18.04 and a 24-core Intel Xeon Gold 5220R (Cascade Lake) processor. Other necessary software dependencies are outlined for each component. The artifacts should produce the graphs shown in the paper as well as reproduce the attack performance.

A.2 Artifact check-list (meta-information)

- **Compilation:** GCC 7.5.0
- **Run-time environment:** These artifacts have been tested on Ubuntu 18.04. The main software dependencies are GCC 7.5.0 and Python ($\geq$ 3.6). Root access is needed to facilitate the reverse-engineering process.
- **Hardware:** Intel Xeon Gold 5220R (Cascade Lake)
- **Run-time state:** The experiments are sensitive to the state of the on-chip network. This means that cache activity (which creates network traffic) can add noise to the experiments.
- **Security, privacy, and ethical concerns:** The experiments in this artifact do not attempt to maliciously exploit any systems.
- **How much disk space required (approximately)?:** 4 GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 48 hours
- **Publicly available (explicitly provide evolving version reference)?:** `https://github.com/CSAIL-Arch-Sec/dont-mesh-around`
- **Code licenses (if publicly available)?:** MIT License
- **Archived (explicitly provide DOI or stable reference)?:** `https://github.com/CSAIL-Arch-Sec/dont-mesh-around/releases/tag/usenix2022`

A.3 Description

A.3.1 How to access

The artifact can be downloaded by cloning our Github repository.

A.3.2 Hardware dependencies

These artifacts target the Intel Xeon Gold 5220R (Cascade Lake) processor. In particular, the processor must be an Intel Skylake SP or Cascade Lake processor which use the mesh interconnect. Evaluating the artifact should require less than 4 GB of disk space.

A.3.3 Software dependencies

These artifacts were tested on Ubuntu 18.04. The software requires Python ($\geq$ 3.6) and GCC 7.5.0.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

None of the experiments in the artifact attempt to maliciously exploit any systems. However, they require access to a server-class processor that is normally shared by many users and will change some system configurations. Be sure to check the original values of the configurations modified in the setup script and restore them afterwards.

A.4 Installation

All installation instructions are included in the artifact. Users should be comfortable using APT and Python/Pip to install dependencies. Familiarity with Git is needed to clone the repository. Additionally, some experiments are long-running and should be run inside a `tmux` session. More specialized experiments come with guidance on how to set them up.

A.5 Experiment workflow

Each experiment is contained in its own directory and is built with its own Makefile. A README within each directory contains detailed information on how to build and run the experiment. It is recommended that the experiments be run in the order presented.

A.6 Evaluation and expected results

Our paper makes the following claims:

1. We can reverse-engineer previously-unknown details about Intel's mesh interconnect.
2. The reverse-engineering results can inform the construction of covert and side-channel attacks.
3. The reverse-engineering results allow for the construction of an analytical model that accurately predicts interconnect-based leakage.
4. The analytical model offer insights into mitigating interconnect-based side channel attacks.

The key results of our paper are detailed below. Detailed instructions on how to reproduce each key result are included in the artifact.

A.6.1 NoC Reverse-Engineering

We reverse-engineered the lane-scheduling policy and priority arbitration policies that dictate how traffic flows on the interconnect. These policies are not specified by Intel and have not been publicly reverse-engineered prior to our paper but are critical to understanding the precise conditions necessary to generate contention. We verify these results by reproducing the two case studies shown in the paper.

A.6.2 Covert Channel

In this section, we demonstrate a working covert channel using only contention on the interconnect that can achieve a capacity comparable to that of previous interconnect-based covert channels (1.5 Mbps $\pm$ 0.3). Our artifact can reproduce the latency trace and capacity plot shown in the paper.

A.6.3 Side Channel

Secret keys can be extracted from vulnerable ECDSA and RSA implementations via the interconnect channel. Single-bits can be classified with an accuracy of at least 69% and 71% respectively and full keys can be recovered with majority voting.

A.6.4 Analytical Model

The reverse-engineering results can be used to construct an analytical model of network contention that accurately predicts observed results. The analytical model can be used to create non-invasive mitigations that reduce the effectiveness of our side-channel. These artifacts should be able to reproduce Figures 12, 13, and 14 in the paper.

A.7 Experiment customization

Running the experiments in the artifact requires first reverse engineering the layout of the tiles on the die, including where the partially and fully disabled tiles are. Because the location of these disabled tiles may differ between different units of the same processor, this must be done before attempting to run the experiments on a different machine. Guidelines for how to do this reverse-engineering are described in Section 3 and Appendix B of the paper.

A.8 Notes

N/A

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact provides the code for the tool and the experiments, as described in the paper. The artifact describes two workflows. The first workflow, WEBGRAPH, runs the classification pipeline: we crawl a set websites, build their graph representations and extract features from these representations, and train a classifier to detect advertising and tracking services (ATS) on the sites. The second workflow, Robustness, consists of experiments to perform content and structural mutations on graph representations to evade a classifier. The artifact consists of three components. First, the source code in a GitHub repository that allows a user to set up and run the described workflows from scratch. Second, two docker images with all dependencies installed, one with just the code and the other with a sample database of 100 crawled sites to test the pipelines. Three, a Google Drive folder with datasets from a larger crawl of 10,000 sites, to validate the performance of the classifier and use for other experiments. All the steps required to run and evaluate the pipeline are described in this document and the repository READMEs.

A.2 Artifact check-list (meta-information)

- **Program:** WEBGRAPH (sources included), Forked OpenWPM (link to repo included)
- **Data set:** Sample crawl of 100 sites to test the pipeline. SQLite and LDB databases (included in the docker image). Google Drive link to datasets form a 10k website crawl for further evaluation.
- **Run-time environment:** The project has been run and tested on Ubuntu 18.04. For the setup, you need to have python3, miniconda, binutils, pip, gcc and g++ installed. All requirements are outlined in the repository. We also provide a Docker image with all the dependencies installed.
- **Hardware:** Having a pod on a Kubernetes cluster is preferable but not necessary due to necessity to stay running for long periods of time.
- **Execution:** During the crawl, the user should use the openwpm virtual environment created during the installation of OpenWPM.
- **Metrics:** Classification metrics (accuracy / precision / recall /F1-score / feature importances).
- **Output:** The crawl outputs SQLite and LDB databases. The WEBGRAPH pipeline takes in the crawl output, and outputs three csv files: graph.csv, features.csv and labelled.csv (the details of the files are outlined in the code-base). The classification task outputs a log for the metric statistics in a report file. The robustness evaluation task generates CSV graph files that can be fed into a trained model, and information about classification switches caused by the mutation.
- **Experiments:** Scripts and instructions to fully reproduce the paper's results are provided in the artifact README files.
- **How much disk space required (approximately)?:** The code-base size is around 18.5 MB. The output size varies depending on the number of websites crawled (around 3-4 MB per website on average). The input database size varies also depending on the number of crawls (around 4.5 MB per website on average).
- **How much time is needed to prepare workflow (approximately)?:** The setup of the repository code and the environment should take around 20 to 45 minutes. The crawling time depends on the number of websites (from hours to days). In case you want to accelerate the setup time, you can use the provided docker image with a pre-built project.
- **How much time is needed to complete experiments (approximately)?:** The total pipeline time depends on the number of websites being analyzed (hours to days approximately). Our tests show that on average, WEBGRAPH takes 0.72 seconds to build the graph, 15 seconds to extract features, and 0.25 seconds to train and test each website. The structure robustness experiments use the WEBGRAPH workflow, but perform graph building and training at every iteration, so the time increases accordingly.
- **Publicly available?:** Yes, on https://github.com/spring-epfl/WebGraph
- **Code licenses (if publicly available)?:** MIT
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/spring-epfl/WebGraph/releases/tag/usenix-artifacts-final

A.3 Description

A.3.1 How to access

The source code is available as a stable tag on GitHub at the following URL: https://github.com/spring-epfl/WebGraph/releases/tag/usenix-artifacts-final. To access it, you can either download the zipped source code or clone the repository.

We also provide two docker images with all the dependencies installed to avoid the setup phase. The image details are as follows:

- **WEBGRAPH image:** Available at https://hub.docker.com/r/springepfl/webgraph. This image contains all the dependencies and the code, and can be used to run the entire pipeline.

- **WEBGRAPH-demo image:** https://hub.docker.com/r/springepfl/webgraph-demo. In addition to all the dependencies and code, this image contains a database of 100 sites (crawled using OpenWPM). The image can be used to work with some test data.

Finally, we provide a Google Drive link with a dataset from a large crawl of 10k websites to evaluate the classifier. The dataset consist of features and labels for different feature configurations of AdGraph and WEBGRAPH. The dataset can be used to replicate all the results of Table 2 in the paper. Link to the dataset: https://drive.google.com/drive/folders/1nDH74p9tLVLvm62DfrsxcO7mraWAWiZa?usp=sharing.

A.3.2 Hardware dependencies

The code is meant to be run on Ubuntu 18.04 with the dependencies mentioned in section A.2. We recommend running the code on a server instance with a fast access to the Internet to accelerate computations. The code-base size is around 18.5 MB. The dataset size can span from 400 MB to several GB depending on the number of crawled websites. We provide a sample dataset of around 100 sites (400 MB).

A.3.3 Software dependencies

- **Custom OpenWPM:** If you intend to run crawls on your own, you need to download a custom OpenWPM tool from this URL: https://github.com/sandrasiby/OpenWPM/tree/webgraph. The installation instructions can be found in the README section of the repository. A sample crawl is included in the WEBGRAPH code-base; you can copy it and run it in the OpenWPM code base. Further instructions are included in the README section of the repository. For convenience, OpenWPM is also installed in the WEBGRAPH Docker image.

- **Docker:** If you opt to use the Docker image, you need to install Docker. After that, you can launch a Docker container using the image and run the experiments in the container.

A.3.4 Data sets

We include a sample dataset in the artifact. Additionally, you can crawl your own dataset using the custom OpenWPM tool following the instructions in the README.

A.3.5 Models

While we do not include a model in the artifact, the dataset on Google Drive can be used to build a model.

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

- **Installing locally on Ubuntu:** Initially you need to setup the environment as described above. Next, follow the instructions in the OpenWPM README to download and install the OpenWPM tool-set. Finally, download the WEBGRAPH code-base and follow the instructions in the README to setup the project and run the various evaluation tasks.

- **Using the Docker container:** We provide a pre-built Docker image that you can load in a docker environment and start running the experiments immediately.

A.5 Experiment workflow

There are two workflows in the artifact. The main workflow is the WEBGRAPH process. This process consists of crawling sites, building their graph representation, and training a classifier based on features extracted from the graph representations. The second workflow handles the robustness experiments (Section 3 and Section 5 of the paper). This workflow uses the graphs generated by the WEBGRAPH process, and performs different types of mutations in order to evade the classifier. We perform two types of mutations – content mutation (Section 3 in paper) and structure mutation (Section 5 in paper). We describe the workflows below:

- **WEBGRAPH:**

  1. Gather crawl data, either using OpenWPM, or the sample database we provide: To run a crawl with OpenWPM, follow the instructions on the WEBGRAPH repository README to install and activate the environment for OpenWPM. Then, update the script `demo.py` in the OpenWPM codebase to feed in the list of sites you want to crawl, and run the script. The crawl process results in a `datadir` directory containing the output files of the crawl.
  2. Build graph representations, and extract features and labels: Edit the file in the WEBGRAPH repository, `features.yaml` to select the feature set that you want to extract. Run the script `code/run.py`. The README contains information on the arguments accepted by the script. This script first reads in the OpenWPM output files and creates graph

representations of each site that we crawled. It then extracts the feature representations based on the parameters provided in `features.yaml`. Then , it extracts labels for each node in the graph representation using filter-lists. The output of the script is three CSV files: a graph representation file, a features file, a labels file.

3. Train the classifier based on the features and labels files, and get the classifier evaluation reports. Run the classification process (`code/classification/classify.py`) to perform 10-fold cross-validation. This consists of classifier metrics (accuracy, precision, recall), the ground truth and predicted labels of the classifier, and feature importances.

- **Robustness (content mutation):**

1. Run the WEBGRAPH workflow to obtain graph files and classifier predictions. Use the argument `--save_model` when running classification so as to have a trained model against which the mutation attacks can be run.
2. For the sites that you want to perform content mutation on, run `code/run.py` to generate the graphs/features/labels.
3. Get original classifier predictions by running `code/classification/classify_with_model.py`
4. Run `robustness/content_mutation/content_mutation.py`. The README in the folder describes the inputs required for the script. Running the script yields a new graph file with content mutation applied on the graph nodes. This can be fed as input to feature extraction and labelling (`code/run_extraction.py`) and then to the trained classifier model to evaluate performance on mutated data.
5. Structure mutation will yield an output file indicating how many classifier predictions (on adversarial and non-adversarial nodes) switched as a result of the mutation. This file helps analyze the impact of the mutation on the adversary's performance.

- **Robustness (structure mutation):**

1. Run the WEBGRAPH workflow to obtain graph files and classifier predictions. Use the argument `--save_model` when running classification so as to have a trained model against which the mutation attacks can be run.
2. Run `robustness/structure_mutation/greedy_mutation.py` (the README in the folder provides the instructions on what to adjust in the config file for the script). This will perform the structure mutation.
3. Structure mutation will yield an output file, `diff_stats`, indicating how many classifier predictions (on adversarial and non-adversarial nodes) switched as a result of the mutation. This file helps analyze the impact of the mutation on the adversary's performance. The output file called `overall_stats` gives you an overview of how many nodes originally had to be flipped. The success rate and the collateral damage can be calculated from these output files, as described in Sec 5.3 (page 10/11) of the paper.

We provide a detailed explanation of how to run these workflows in the repository READMEs.

A.6 Evaluation and expected results

The main tool in the paper is WEBGRAPH, which classifies URLs on sites as advertising and tracking (ATS) or benign (non-ATS). WEBGRAPH performs comparably to other classifiers despite not using brittle content features, due to the addition of a new set of features, *flow*, based on ATS behavior. Our artifact enables a user to run the WEBGRAPH pipeline: from the crawl to the graph creation and feature extraction to the classifier training. We provide a test dataset of 100 crawled sites to analyze how the pipeline works. At the same time, this test dataset is too small to validate the performance of the classifier. In order to facilitate verification of WEBGRAPH's performance, we also provide feature and labels from a crawl of 10,000 sites. These can be fed into the classifier to obtain results, and used to train a model that can be used in further experiments (such as the robustness workflow). We opt to provide the processed features and labels for two reasons. First, the raw crawl database of 10,000 sites would be large (in the order of several GB) without necessarily providing much value for evaluation. Second, the processed features and labels can be used for other experiments (an evaluator can use as many or as few sites as they desire to train the model). We note that these files themselves are $\approx$ 600MB. The datasets provided can also be used to replicate the results shown in Table 2 of the paper, but running the classification process (Step 3 of WEBGRAPH workflow) with the feature and labels file.

The paper performs many experiments related to content and structure mutations. The artifact, therefore, also provides code to generate these mutations. A user can run the mutations on graphs generated from either their own crawled data, or on the data that we provide. The READMEs in the robustness sections of the artifact include information on what the expected output of the mutations are. The code also allows a user to run the entire pipeline in two modes: WEBGRAPH and AdGraph. The content mutation robustness workflow, run

with AdGraph mode, can be used to replicate the results of Section 3. The content and structure mutation workflows, run with WEBGRAPH mode, can be used to generate the results of Section 5. Note that in order to generate the results close to the values in the paper, you would have to run crawls with 10,000 sites (for the content mutation experiments) and 100 sites (for the structure mutations experiments).

A.7 Experiment customization

The workflows can be customized as follows:

- **WEBGRAPH:**
    1. The script to run the experiment, `code/run.py`, takes in an argument, `--mode`, which allows you to specify the system you want to run: AdGraph or WEBGRAPH.
    2. The feature extraction process can be modified to use different categories and types of features, as required. In the default version, we do not use content features (which are used in other tools such as AdGraph), but content features can be extracted by modifying `code/features.yaml`. New features can also be added to the classifier.
    3. The labelling process can be modified to include additional filter lists.
- **Robustness:**
    1. The content mutation process can be modified to mutate the URLs in different ways. The process also offers the option to perform the two scenarios described in Section 3 (third party random mutation, and third party as a subdomain of the first party).
    2. The structure mutation process currently offers four types of mutations that the user can choose from. This can be updated as required.
    3. The structure mutation process currently calculates desired and undesired switches for an adversary as described in the paper. This definition can be modified in the code to account for other adversarial goals.

We provide descriptions of the various customization parameters in the repository READMEs.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



C Artifact Appendix

C.1 Abstract

Our work in this paper consists of four separate components:

1. **The cookie consent web crawlers.** The web crawler component uses a series of Python scripts and the OpenWPM framework to gather browser cookies and associated purpose categories from websites. The output of this component is a dataset of browser cookies including category labels.
2. **The feature extraction and XGBoost classifier.** This component uses the collected dataset of cookies and transforms it into a sparse matrix representation, using all properties of a browser cookie in the process. This sparse matrix, combined with the category labels, is then used to train a decision tree model using the XGBoost algorithm. This allows us to predict purpose categories for previously unseen cookies.
3. **The GDPR violation detection scripts.** Using knowledge of the articles of the GDPR and the cookie dataset collected by the consent web crawler, these scripts identify potential GDPR violations on websites in the wild. The output of this component is a dataset of statistics detailing the prevalence of potential GDPR violations, based on 8 different methods of analysis.
4. **The "CookieBlock" browser extension.** This addon provides a privacy protection mechanism for users which automatically deletes cookies that they did not consent to. The extension uses the classifier as the central engine to decide which cookie belongs to what category. It supports Chromium-based browsers as well as Firefox.

The components have been constructed using Python 3 and JavaScript. The webcrawler in particular is based on the OpenWPM framework version 0.12.0, and must be run on Linux. For this reason, we provide an Ubuntu VM that comes with all dependencies preinstalled. We also provide a precomputed dataset of statistics and metrics which stem from our previous executions of these components, and are the datasets used for the results presented in the paper. This includes the candidate domains used for the web crawl, the complete set of performance metrics for the XGBoost classifier and the Cookiepedia baseline, as well as all statistics and data on the GDPR Violation Detection.

No specialized hardware is required to reproduce the results of the paper, but at least 8GB of RAM and 40 GB of disk space are needed. Due to the nature of the dataset collection, results may differ significantly if reproduced at a later date. Instructions on how to compare and validate the results are provided in the form of a detailed “README” document, containing a step-by-step guide detailing each part of the process. Said document also provides links to the source code release for each component.

C.2 Artifact check-list (meta-information)

- **Binary:** Cross-platform virtual machine image, containing all program components and datasets.
- **Data set:** Yes, included. The data set and VM are found at: https://doi.org/10.5281/zenodo.5838646
- **Run-time environment:** The OpenWPM crawler only runs on Linux. The other scripts and the browser extension work on Windows and Linux. An Ubuntu VM image is included.
- **Hardware:** At least 8GB of RAM needed, and approximately 40 GB of disk space. Additional CPU cores can speed up the computation, but works with a single core also.
- **Run-time state:** The results are dependent on the website content, as well as the CMP implementations, which may change over time, and are out of our control.
- **Execution:** With the complete input dataset, the web crawls alone may take between 1 and 2 weeks to complete. With a reduced dataset, the full process takes a few hours.
- **Metrics:** Accuracy, precision, recall, macro-precision, macro-recall and F1 score.
- **Output:** Printed to the console, stored in SQLite databases, JSON and log files. Expected results are included for each step of the process.
- **Experiments:** Collection of the browser cookie dataset, the training and evaluation of the classifier, the GDPR Violation detection and the generation of the extension’s classifier model can all be replicated using commands manually input by the user. We provide a detailed step-by-step guide on the process.
- **How much disk space required (approximately)?:** At least 40 GB is required for the VM. While the included datasets are much smaller than this, the data that is collected and generated may quickly take up disk space.
- **How much time is needed to prepare workflow (approximately)?:** When installing the VM image, only a few minutes. When setting the scripts up natively, at most an hour.
- **How much time is needed to complete experiments (approximately)?:** A few hours.
- **Publicly available?:** Yes, all components are publicly available on Github. Links are provided in the step-by-step guide.
- **Code licenses (if publicly available)?:** The OpenWPM crawler is GPL3 licensed. Other components are MIT licensed.
- **Data licenses:** CC by 4.0 International
- **Archived:** Yes, available at: https://doi.org/10.5281/zenodo.5838646

C.3 Description

C.3.1 How to access

The artifact is publicly available and can be downloaded as a self-contained package from:
https://doi.org/10.5281/zenodo.5838646
It includes a VM image that has all components preinstalled, as well as a README that guides the user to replicate and

reproduce the results. The document also contains links to the original repositories, should the user intend to install the scripts natively.

C.3.2 Hardware dependencies

The artifact requires no specialized hardware to run. A single core machine with 8GB of RAM and mor than 40 GB of disk space should be enough. The VM requires considerable size when set up, which is due to the libraries that are used, and because of the data collection that needs to be performed to replicate the results.

C.3.3 Software dependencies

If the VM image is used, only a virtualization product such as VirtualBox or VMWare is required. All other components should be ready to use. For native installations, some Python and Node libraries are required. The exact details are provided within the step-by-step guide included as part of the artifact.

C.4 Installation

The recommended method of setting up the artifact is to load the virtual machine image using VirtualBox. All further steps are documented in great detail within the README file of the artifact. In the interest of space, we will not repeat the steps here, and instead refer to the README.

C.5 Evaluation and expected results

First, we crawled 6M domains from a Tranco list collected on May 5th. Out of these, 30k were found to have the selected CMPs on them. From these websites, we collected a ground truth of 304k cookies with labels, which we used to train an XGBoost model with 84.4% weighted accuracy. In an analysis of the 30k websites, we found that a vast majority, namely 94.7% of them, contain at least one potential privacy violation. All the steps to reproduce these results together with the intermediate files of our results are documented in great detail within the README file of the artifact.

Note that the changes to websites content cause variance in the results. We try to document this variance below:

1. **Variance for the cookie consent web crawlers.** Within the large Tranco list, the number of websites with CMPs remains roughly the same over time. Among the more popular sites, the percentage of websites using the selected CMPs is higher, allowing the use of smaller input files. In the paper, we observed suitable CMPs on 0.63% of the Tranco 6M list (see Sections 2.1 and 2.2 of the paper). In the Master Thesis report, it was 1.6% for Tranco 1M Worldwide or 1.25% for Tranco Europe, and BuiltWith website reports the selected CMPs in over 3% of the top 1M websites. We observed on average 22 cookies with label per website, which depends strongly on the number of sub-pages visited for each site (discussed in the par. 3 of Section 2.3. of the paper). We did not measure the variance for the settings in the crawler, but the results should be consistent as long as you run the provided crawler from within EU.
2. **Variance in the XGBoost classifier.** The feature extraction is deterministic, extracting the same features with each execution. Training the model appears to be stable, as we observe a standard deviation of 0.23% in the accuracy. The model's balanced accuracy will drop from the reported 84.4% if you use a smaller training dataset. Additional standard deviations for each metric are provided in the dataset.
3. **Variance in the GDPR violation detection scripts.** The observed violations depends on website selection, but the results between the master thesis report and the paper varied by 4% for the number of websites with at least one type of violation. For individual violations this variance can be higher.



A Artifact Appendix

A.1 Abstract

We propose KHALEESI, a machine learning (ML) approach that captures the essential sequential context needed to effectively detect advertising and tracking request chains. We release KHALEESI's classification code, ML model, browser extension, and data sets. Classification code is written in Python 3.6, the ML model is trained using Scikit, the browser extension is written in JavaScript/HTML, and the data is crawled using OpenWPM.

A.2 Artifact check-list (meta-information)

- **Binary:** A browser extension to block advertising and tracking request chains. The extension is designed and tested in Mozilla Firefox.
- **Model:** ML model to detect advertising and tracking request chains. Released ML model was trained on request chains from homepages of Alexa top-10K websites.
- **Data set:** Data sets to train and test ML model. We release crawls of homepages, home and sub pages, home and sub pages with cookies blocked, and home and sub pages with browser spoofed as Safari. All data sets are crawls of Alexa top-10K websites. The data contains requests, responses, and JS execution.
- **Run-time environment:** Scripts can be run using Python 3.6 and above. The code was tested on Ubuntu 16.04.7 LTS.
- **How much disk space required (approximately)?:** We recommend a disk space of ∼100GB to train the classifier. The browser extension does not have any disk space constraints.
- **How much time is needed to complete experiments (approximately)?:** The classifier can be trained in ∼ 10 hours. The browser extension blocks the ads instantaneously.
- **Publicly available (explicitly provide evolving version reference)?:** KHALEESI's code, data, and browser extension is available at https://uiowa-irl.github.io/Khaleesi/.
- **Archived (explicitly provide DOI or stable reference)?:** KHALEESI's code, data, and browser extension is available at https://github.com/uiowa-irl/Khaleesi/tree/bd28513878a363b39b0ee9e7a6a4350f71672912

A.3 Description

A.3.1 How to access

KHALEESI's code, ML model, and browser extension are available on Github at: https://uiowa-irl.github.io/Khaleesi/. Data sets are available on Zenodo at: https://doi.org/10.5281/zenodo.6084582.

A.3.2 Hardware dependencies

KHALEESI ML model was trained on a machine with 16 cores and 96 GB RAM. We recommend a disk space of ∼100 GB to train the classifier. The model can be tested on hardware with less resources.

A.3.3 Software dependencies

KHALEESI browser extension was designed and tested on Mozilla Firefox. We trained and tested KHALEESI ML model on Ubuntu 16.04.7 LTS.

A.3.4 Data set dependencies

KHALEESI is trained on data set crawled through OpenWPM version 0.10.0. The code might require some minor modifications to process data from newer versions of OpenWPM.

A.4 Installation

We provided instructions to run KHALEESI on Github.

A.5 Experiment workflow

In addition to instructions on Github, we provide detailed instructions to run the code below:

A.5.1 Training & Testing ML model

We list the step-by-step process to train and test KHALEESI's ML model below:

1. *Data collection:* Collect network and JavaScript initiated requests using OpenWPM.
2. *Request chain construction:* Organize network and JavaScript initiated requests into chains. Request chains can be constructed with HTTP and JavaScript chain construction scripts.
3. *Request chain labeling:* Once constructed, label request chains using EasyList (EL) and EasyPrivacy (EP) filter lists. Use filter list labeling script and EL/EP filter lists to label the chains.
4. *Feature extraction and transformation:* After labeling, extract features from the request chains using using feature extraction script and encode them using feature encoding script.
5. *Model training:* Since, KHALEESI relies on previous confidence as a feature, extract the previous confidence for each request in a chain before training the final model. The previous confidence can be extracted using compute previous confidence script. The last block of previous confidence script stores the final trained model. An already trained model is available in data directory.
6. *Testing the model:* KHALEESI uses 10-fold cross validation to test the data sets. The encoded features with previous confidence can be tested using test classifier script and the accuracy can be computed using compute accuracy script.

A.5.2 Analysis of Request Chains

We release scripts to analyze cookie syncing and bounce tracking instances in request chains. Use the cookie syncing and bounce tracking scripts to identify cookie syncing and bounce tracking instances, respectively.

A.5.3 Browser Extension

To add KHALEESI to Firefox, enter `about:debugging` in the URL bar, click *This Firefox*, click *Load Temporary Add-on*, navigate to the extension's directory and open `manifest.json`. To view the requests blocked by KHALEESI, open extension's console by clicking *Inspect* in `about:debugging` or see the network tab in the Firefox Developer Tools.

A.6 Evaluation and expected results

Training & Testing ML Model: Upon successful execution, the workflow should produce a trained ML model and output its accuracy.

Analysis of Request Chains: Upon successful execution, the scripts should list the cookie syncing and bounce tracking instances in request chains.

Browser Extension: After installation, the browser extension should block advertising and tracking request chains.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact provides the steps for demonstrating the functionality of our system, minTAP, and reproducing the main results in the paper. It includes proof-of-concept implementations of both the client and the server based on the minTAP protocol. The client is implemented as a Chrome extension, while the server hosts a minTAP-compatible service in a docker container.

A.2 Artifact check-list (meta-information)

- **Data set:** We use a non-public dataset that takes about 300 MB.
- **Run-time environment:** The client requires a Chrome browser with developer mode enabled. The server requires Ubuntu 20.04 with Docker installed.
- **Metrics:** Privacy benefits (in terms of data minimized) and execution time.
- **Output:** Graph and console outputs that should closely match the results given in the original paper.
- **Experiments:** this artifact consists of three experiments: verifying minTAP functionality, replicating privacy benefits, and replicating execution time.
- **How much disk space required (approximately)?:** 500 MB.
- **How much time is needed to prepare workflow (approximately)?:** 1-2 hours.
- **How much time is needed to complete experiments (approximately)?:** 1-2 hours.
- **Publicly available (explicitly provide evolving version reference)?:** The code will be publicly available in https://github.com/EarlMadSec/minTAP.
- **Archived (explicitly provide DOI or stable reference)?:** https://doi.org/10.5281/zenodo.6523010.

A.3 Description

A.3.1 How to access

An archived version of the code is available at https://doi.org/10.5281/zenodo.6523010.

A.3.2 Hardware dependencies

We use an AWS EC2 `t3.large` instance, but any machine with similar hardware specifications should also work.

A.3.3 Software dependencies

The following are the software dependencies for minTAP. The provided Docker setup file will manage other dependencies.

- **Client:** Chrome 98
- **Server:** Ubuntu 20.04 with Docker 20.10 installed

A.3.4 Data sets

We did not publish the dataset.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Server installation. Please follow the instructions located under `Server/README.md` for building and running the docker container. The docker will set up a minTAP-compatible service on the host machine's port 5000. Make sure both inbound and outbound network traffics are allowed on this port.

Test account setup. You need to create a developer account at https://ifttt.com/developers and follow the steps below to register the minTAP-compatible service with IFTTT.

1. Create a new service named `mintap_service` in https://ifttt.com/services/new and add a new trigger based on the instructions in `Server/README.md`.
2. Go to https://platform.ifttt.com/services/mintap_service/api and fill the `IFTTT API URL` field with the URL path to the minTAP-compatible service (i.e., port 5000 on the server host machine).
3. Go to https://platform.ifttt.com/services/mintap_service/api/authentication and fill the `Authorization URL` field with `[IFTTT API URL]/mintap/auth/authorize` and the `Token URL` field with `[IFTTT API URL]/mintap/auth/token`.

Once the server information is set up, go to the following links to run the IFTTT's built-in sanity checks. If the server is set up correctly, all tests should pass.

- https://platform.ifttt.com/services/mintap_service/api/endpoint_tests
- https://platform.ifttt.com/services/mintap_service/api/authentication_test

Client installation. Please follow *Step 2* in this Chrome help page to install minTAP's browser extension. The client's source code is located inside the `Client/` folder. Note that the client does not need to be installed on the same machine as the server.

A.5 Experiment workflow

The experiment comprises 3 components:

1. **Functionality test.** The first workflow shows how our minTAP client and server integrate with IFTTT. It requires using the test account to create and modify rules in IFTTT. Click the `Personal Applets` link in IFTTT Developer Dashboard and create a new rule that uses `mintap_service` as the trigger service. You may modify the filter code inside the rule to change its behavior. Once you hover over the save button, minTAP's client will automatically fill the minTAP's related fields. See the Usage section in `Server/README.md` on how to manually trigger the rule.

2. **Analysis of privacy benefit.** The second workflow describes the procedure to reproduce the analysis in Section 7.2. Refer to the description under `rule_analysis/README.md` for detailed instructions. Due to potential privacy concerns, we did not publish the original dataset we used in the paper.

3. **Execution time.** We provide a test API to measure the latency overhead of minTAP in terms of the execution time of its additional operations. The test API can be accessed using the curl command:

   ```
   $ curl "[IFTTT API URL]/ifttt/v1/triggers/bench"
   ```

A.6 Evaluation and expected results

Following is a description of the expected results after running each workflow.

1. **Functionality test.** After the rule is run, check the rule's activity logs on IFTTT's website (by clicking the `view activity` button in the rule's page) to confirm that all unneeded trigger attributes are removed.

2. **Analysis of privacy benefit.** A plot similar to Figure 9 should be generated.

3. **Execution time.** The test API will return the average latency overhead (in seconds) over 20 runs. The value should be less than 0.03 seconds.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

In this evaluation, we will allow you to run our experiments on our verifiable secret sharing schemes. One scheme is KZG based with trusted setup, and the one is Virgo based without trusted setup. We will use the c++ standard library chrono to time our execution. You need to read about 20 *lines of C++ code to verify our time measurement.*

A.2 Artifact check-list (meta-information)

- **Algorithm: KZG polynomial commitment, Virgo zero-knowledge proofs protocol, FFT**
- **Program: C++ program**
- **Compilation: cmake, and we will provide a bash file for fast setup the environment and fast execution.**
- **Run-time environment: Ubuntu**
- **Hardware: Amazon c5a.24xlarge**
- **Execution: We provide a bash file for execution**
- **Metrics: measure time in seconds**
- **Output: the dealer's (prover) execution time and the verifier's execution time**
- **Experiments: our improved KZG execution, and our virgo based VSS execution**
- **How much disk space required (approximately)?: 20GiB for the OS, we do not require any additional space.**
- **How much time is needed to prepare workflow (approximately)?: 1 hour**
- **How much time is needed to complete experiments (approximately)?: 10 hours**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: GPLv3**

A.3 Description

A.3.1 How to access

We put the ssh key in our github repo root directory, named "evss_AE.pem". To access the machine, visit this link:

https://bit.ly/3AFpnwk

If the link fails, use information below:

1. USER NAME: AE_EVSS
2. Password: uNe*g!)0H8pzu=0
3. Access key ID: AKIAWVJ5RUVJDZXGF2X7
4. Secret access key: TcVJ-FocQ/ztXUlYA4HdOu3I9dP/SW8NFo1KuanWu
5. Console login link: https://458079970642.signin.aws.amazon.com/console

With the access link, you can now access our AWS account. At region North California, you should be able to find a machine labeled evss_AE. You can start the machine and find it's IP address. Assuming you have the machien IP, use this command to access:

ssh -i evss_AE.pem ubuntu@IP

A.3.2 Hardware dependencies

Amazon AWS c5a.24xlarge, 20Gib disk space.

A.3.3 Software dependencies

libgmp, ate-pairing, xbyak

All dependencies will be installed via our provided script: dependency.sh

A.4 Installation

Obligatory. Describe the setup procedures for your artifact targeting novice users (even if you use a VM image or access to a remote machine).

In the home directory, you should be able to find a folder named eVSS. This folder contains all needed files to run the experiment. Dependencies are pre-installed.

A.5 Experiment workflow

The machine should be ready to directly run the experiment. At the eVSS directory, you can run "./compile.sh" to compile the whole project. (It's already pre-compiled for you, but in case you want to compile it, you can run this command.)

Then to run the experiment for improved KZG-based VSS, run "./trusted_setup_version.sh"

To run the experiment for transparent VSS, run "./transparent_version.sh"

We will only run experiment for verifiable secret sharing. Let t_p be the VSS's prover time, and t_v be the VSS's verification time, you can calculate the DKG time for n parties by the following formula:

$$\mathsf{DKGtime} = (2n \times t_v) + 2 \times t_p$$

A.6 Evaluation and expected results

1. For the KZG-based VSS, we claim:
   (a) the running time for 2^{10} players is 3 second
   (b) the running time for 2^{15} players is 100 second
   (c) the running time for 2^{20} players is 4000 second
   (d) the verification time is constant, 0.001 second
   (e) the proof size is constant, 192 Byte
2. For the transparent VSS, we claim:
   (a) the running time for 2^{10} players is 0.2 second, proof size 223840 Bytes, and verification time 0.003 second.
   (b) the running time for 2^{15} players is 8 second, proof size 324832 Bytes, and verification time 0.004 second.
   (c) the running time for 2^{20} players is 300 second, proof size 467520 Bytes, and verification time 0.01 second.

A.7 Note

For final stable URL, visit

https://github.com/sunblaze-ucb/eVSS/tree/e8f1cd4d6ef086b2ae017ed56560328fdffec491





A Artifact Appendix

A.1 Abstract

This artifact contains a reference implementation of the ppSAT protocol, as well as supporting benchmarks and helper scripts necessary for reproducing our experimental results. Our implementation uses the EMP-toolkit framework for the underlying secure computation. Specifically, the artifact contains an implementation of the protocol as a secure distributed program, compiling which produces a ppSAT solver binary. This binary is then used by our testing infrastructure, and would in production be distributed amongst the parties and invoked over their local, private formulas to execute the SAT solving decision procedure over their conjunction. The artifact also includes functionality to support running microbenchmarks on individual giant steps of the ppSAT protocol over random formulas, as well as our haplotype inference benchmarks (and supporting scripts). It further includes code to execute the ppSAT protocol 'in the clear' to allow evaluating the overhead of the secure computation.

The artifact is composed of C++ code supported by Python scripts, intended for execution on suitable x86-based hardware running Ubuntu 20.04 or a similar, modern desktop Linux distribution. The compilation and benchmarking are verified by the Github Action CI.

A.2 Artifact check-list (meta-information)

- **Algorithm:** the ppSAT solver algorithm, including the underlying oblivious stack
- **Program:** implementations of private ppSAT and its tests, as well as of ppSAT 'in the clear'
- **Compilation:** cmake and make
- **Data set:** HapMap dataset, publicly available at https://web.archive.org/web/20170706011547/http://www.stats.ox.ac.uk/~marchini/phaseoff.html
- **Run-time environment:** tested on Ubuntu 20.04, should work on all modern desktop Linux distributions
- **Hardware:** tested on Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz * 6 processor
- **Execution:** shell scripts, python scripts, x86 binary
- **Security, privacy, and ethical concerns:** the HapMap dataset is publicly released and widely used in genetic studies
- **Metrics:** running time
- **Output:** the running time and model (or UNSAT) for satisfiable formulas (or unsatisfiable formulas)
- **Experiments:** all experiments are verified by a Github Action composed of four items: `functionalities` (unit testing of our protocol components); `microevaluation` (benchmarking of our protocol components, produces figures in §5.1); `simulation_difference` (used to verify that our estimated time is close to wallclock running time, as mentioned in §5.2); and `benchmark` (reproduces Sections 5.2 and 5.3 – we only include one test because running all benchmarks exceeds 6 hours of running time)
- **How much disk space required (approximately)?:** 10 GB
- **How much time is needed to prepare workflow (approximately)?:** 2 hours
- **How much time is needed to complete experiments (approximately)?:** 4 hours for benchmarking our cryptographic protocols; 12 hours for obtaining the number of steps that our solver needs for Haplotype benchmarks
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/PP-FM/ppsat
- **Archived URL :** https://github.com/PP-FM/ppsat/releases/tag/v1.0.0

A.3 Description

A.3.1 How to access

https://github.com/PP-FM/ppsat

A.3.2 Hardware dependencies

A modern x86 CPU.

A.3.3 Software dependencies

cmake, emp-toolkit, gtest, openssl

A.3.4 Data sets

The HapMap dataset. Our repo includes the data that are used for our benchmarking.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

We use a publicly released, widely-used dataset.

A.4 Installation

Installation can be easily done by following the instructions at https://github.com/PP-FM/ppsat#installation. Our Github Action scripts contain all steps to install our code on a clean Ubuntu machine.

A.5 Evaluation and expected results

The latest Github Action results, at the point of submission, can be found at https://github.com/PP-FM/ppsat/actions/runs/1894455944. The output of each subtask includes the running time and status of each test. They can be used to plot the figures presented in the paper.

In the paper, we made the following claims.

1. Our ppSAT solver can correctly and reasonably efficiently solve SAT formulas based on our newly designed heuristics, and all components scale well when the size of the formula increases (§5.1). This claim is tested in `artifact/functionalites` and `artifact/microevaluation`.

2. Our ppSAT solver can be used towards a real application of solving haplotype inference (§5.2). The accuracy of our timing estimation is tested in `artifact/simulation_difference`, while the rest is benchmarked in `artifact/benchmark`.

3. Our ppSAT solver still incurs a high overhead compared with a plaintext solver (§5.3). The results from §5.3 only require a plaintext SAT solver, for which we use Kissat.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

Hyperproofs artifact contains two components: (1) source code of the Hyperproofs vector commitment (VC) scheme and (2) scripts to compare the performance of Hyperproofs aggregation with SNARKs based Merkle-proof aggregation (implemented by Ozdemir *et al.* [3,4]).

We use the Golang bindings of the `mcl` library [2] to implement Hyperproofs. Hyperproofs source code contains three major components: (1) the vector commitment scheme, (2) implementation of the argument system for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ using the inner-product argument (IPA) proposed by Bünz *et al.* [1], and (3) KZG commitment scheme to optimize the verifier of the IPA.

A.2 Artifact check-list (meta-information)

- **Algorithm:** We implement the Hyperproofs vector commitment scheme described in the paper.
- **Compilation:** Hyperproofs require go 1.16 or above and `mcl` requires GCC 9.3.0 and above. Baseline implementation from Ozdemir *et al.* requires rust [3].
- **Run-time environment:** Ubuntu 20.04 or similar with sudo privileges (for `mcl` installation).
- **Hardware:** Our benchmarks used a machine with Intel Core i7-4770 CPU @ 3.40 GHz with 8 cores and 32 GiB of RAM.
- **Execution:** Benchmarks are single-threaded and memory-intensive thus the benchmarking results can vary due to simultaneous usage of resources by other processes. Approximately, micro and macro benchmarks (excluding Com and OpenAll) take 1.5+ hours, and comparison with SNARKs based Merkle-proof aggregation takes 6.5+ hours.
- **Metrics:** Experiments report the execution time of VC operations.
- **Output:** The artifact returns the execution time of benchmarks reported in the paper.
- **Experiments:** At a high level, we evaluate the performance of our VC scheme through micro-benchmarks, macro-benchmarks, and baseline comparison. Instructions to set up and run the experiments are included in the readme of the corresponding project repositories (see App. A.3.1).
- **How much disk space required (approximately)?:** In total, 150 GiB of storage is required. This is because the public parameters of the vector commitment scheme requires 100 GiB and SNARKs based Merkle-proof aggregation requires 50 GiB of storage.
- **How much time is needed to prepare workflow (approximately)?:**

Step	Estimated time (hours)
Software installation	1+
Generating Hyperproof public parameters	1.5+
Generating SNARK [3] public parameters	8+
Total	10.5+

- **How much time is needed to complete experiments (approximately)?:**

Step	Estimated time (hours)
Benchmark Open, Com	6.5+
Other micro/macro-benchmarks	1.5+
Hyperproofs aggregation	4+
SNARK + Merkle aggregation	2.5+
Total	14.5+

- **Publicly available (explicitly provide evolving version reference)?:** Yes (see App. A.3.1).
- **Code licenses (if publicly available)?:** Apache License, Version 2.0
- **Archived (explicitly provide DOI or stable reference)?:** Yes (see App. A.3.1).

A.3 Description

A.3.1 How to access

The stable URL to access the artifact:

https://github.com/hyperproofs/hyperproofs/releases/tag/1.0.0

The latest version of the artifact is available at:

https://github.com/hyperproofs/hyperproofs/

A.3.2 Hardware dependencies

Requires at least 32 GiB of RAM and 150 GiB of storage.

A.3.3 Software dependencies

Requires Ubuntu 20.04 with sudo privileges, go 1.16 or above, rust nightly, GCC 9.3.0 or above, CMake, libgmp, libflint, git, python3 (pandas and matplotlib), curl, and other standard tools.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

We include the detailed installation instructions in the project repository (see App. A.3.1).

A.5 Experiment workflow

Once the necessary software tools are installed:

- **Setup:** First, run the scripts/hyper-go.sh in hyperproofs-go. This generates the public parameters for the VC scheme, which will be located in the folders pkvk-26 and pkvk-30. Second, run the merkle-snarks-setup.sh script in bellman-bignat. This generates the public parameters for the SNARKs baseline in the folders pedersen-30 and poseidon-30.
- **Benchmarks:** First, run the scripts/hyper-bench.sh in hyperproofs-go. This generates the execution times of various VC operations that constitute micro- and macro-benchmarks reported in the evaluation section of the paper. Moreover, this script also generates the proving and verification times of Hyperproofs aggregation scheme. Second, run the merkle-snarks-bench.sh script in bellman-bignat. This script computes and reports the proving and verification times of SNARK based Merkle-aggregation.

A.6 Evaluation and expected results

The evaluation section of the paper presents: (1) micro-benchmarks, (2) macro-benchmarks, and (3) comparison with SNARK based Merkle-tree aggregation. By running the scripts/hyper-bench.sh, raw data for micro-benchmarks can be obtained. Thus, the micro-benchmarking numbers can be used to directly derive the macro-benchmarks. Additionally, micro-benchmarking script returns the performance of aggregation in Hyperproofs for varying batch sizes. The SNARKs baseline can be obtained by running merkle-snarks-bench.sh.

A.7 Experiment customization

N/A

A.8 Notes

N/A

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

References

[1] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for Inner Pairing Products and Applications. Cryptology ePrint Archive, Report 2019/1177, 2019. https://ia.cr/2019/1177.

[2] Mitsunari Shigeo. mcl: a portable and fast pairing-based cryptography library. https://github.com/herumi/mcl/, 2015. Accessed: 2020-10-14.

[3] Alex Ozdemir. bellman-bignat, 2020. https://github.com/alex-ozdemir/bellman-bignat.

[4] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling Verifiable Computation Using Efficient Set Accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.



A Artifact Appendix

A.1 Abstract

"Loki: Hardening Code Obfuscation Against Automated Attacks" is a paper on code obfuscation that focuses on hardening VM handlers with the goal of thwarting automated attacks such as symbolic execution.

Our artifact includes both the source code of our prototype, which allows to create obfuscated binaries, and the attack tooling used in the evaluation (on Github) as well as the data generated during the evaluation (published as dataset on Zenodo). Our experiments cover various aspects from measuring Loki's overhead to measuring its resilience w.r.t. to automated attacks. All experiments come with a README.md explaining the individual scripts and a wrapper script to improve usability. We use a Docker container to minimize setup problems and make the artifact accessible to different setups.

Evaluating this artifact will require

1. Building a docker container (and potentially downloading up to 50GB of data from an accompanying Zenodo artifact)
2. Creating a number of obfuscated binaries (we provide convenience wrapper scripts doing all the work)
3. Running 14 experiments (most of which have multiple steps):
   - Validating correctness and measuring overhead
   - Running multiple attacks

Individual experiments may run multiple hours (depending on whether (1) you want to use binaries created by us or create them yourself and whether you focus on replicating the results on a subset or intend to test all binaries).

A.2 Artifact check-list (meta-information)

- **Data set: `https://zenodo.org/record/6686932`**
- **Hardware: 52 cores + 64GB RAM + 25GB disk space**
- **Experiments: 14 different ones, covering all aspects**
- **How much disk space required (approximately)?: 25GB**
- **How much time is needed to prepare workflow (approximately)?: 1h**
- **How much time is needed to complete experiments (approximately)?: 40h (experiments can run unattended after being launched)**
- **Publicly available (explicitly provide evolving version reference)?: Yes**
- **Code licenses (if publicly available)?: AGPL 3**
- **Data licenses (if publicly available)?: AGPL 3**
- **Archived (explicitly provide DOI or stable reference)?: 10.5281/zenodo.6686932**

A.3 Description

Our artifact is split into two parts: The core component is the source code of our prototype and evaluation tooling (published on Github). Beyond that, we published artifacts such as produced binaries and raw results in a dataset on Zenodo. In essence, our artifact includes 14 experiments, which create obfuscated binaries, evaluate their overhead/correctness, or attack them using a number of automated simplification attacks.

Our code is intended to be run in a (provided) Docker container.

A.3.1 How to access

- Download the source code from Github: `https://github.com/RUB-SysSec/loki/commit/86134c1318347547debaf9b77e867d5b16d79d1d`
- Download the dataset from Zenodo: `https://zenodo.org/record/6686932`

A.3.2 Hardware dependencies

We recommend a server with many CPU cores (to reduce the experiment runtime and speed-up evaluation); We recommend more than 52 cores, at least 64 GB RAM, and about 25 GB disk space. These are no hard requirements: Less cores may work, but will increase the runtime of all experiments. Internet access is recommended.

A.3.3 Software dependencies

We provide our code in form of a Docker image. We have not tested the artifact on any OS other than Linux; most distributions should work fine. Optimally, your kernel supports Kernel Samepage Merging (https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html).

A.3.4 Data sets

There is a data set containing evaluation results, binaries, and other artifacts resulting from our evaluation on Zenodo at `https://zenodo.org/record/6686932`. It is 4.9GB when zipped and 16GB when unzipped on disk.

A.4 Installation

1. Download the source code from Github:
   `git clone https://github.com/RUB-SysSec/loki.git`
2. Use our wrapper script to build the Docker container:
   `cd loki && ./docker_build.sh`
3. Start the docker container using our wrapper script:
   `./docker_run.sh`

4. Running this script again will connect you to the container:
   `./docker_run.sh`

5. Within the docker container, install Loki and all dependencies:
   `./setup.sh`

Noteworthy, `docker_run.sh` will mount the loki directory within the container as volume: Simplified speaking, anything within the container that is located within `/home/user/loki/` will be available outside the docker container in the `loki` folder. This can be convenient, e. g., if you want to copy the dataset from Zenodo into the Docker container: Simply place it in the `loki/` folder and it will be accessible from within the container. A more detailed explanation can be found in the Github README.md.

A.5 Experiment workflow

All experiments are located in `loki/experiments/` with the experiment number matching the one in the paper. Each experiment is documented in a `README.md` and usually consists of two to four steps. For your convenience, we provide Python scripts automating that part (`experiment_N.py`). As some experiments can share data (such as binaries generated), we recommend you do not change the default paths suggested (data will almost always be placed in `/home/user/evaluation`). Experiments can be customized by setting command line flags, changing values of globals in the Python wrapper script, or patching the scripts themselves (which we don't recommend generally). Our experiments usually run at least hours up to days (the standard timeout used is always 1 hour for each task; oftentimes there are at least $1,000$ tasks per experiment. We suggest you test the experiment on a subset of tasks (e.g., by generating only 10 binaries instead of $1,000$). **All experiment scripts already propose a more sensible value of tasks.** If this is not desired, changing `NUM_INSTANCES` (or similar-named constants at top of the scripts) allows you fine-granular control of how much tasks are executed.

A.6 Evaluation and expected results

Our experiments cover all relevant aspects. A detailed approach on how to reproduce them can be found in the `README.md` files we provide for each experiment. The expectations of the experiment are outlined in the paper.

- Dead code elimination: Only a few instructions (1-2%) of Loki's handler can be removed
- Experiment 1 Correctness: The obfuscated binaries produced by Loki maintain the same functionality
- Experiment 2 Coverage: Full code and path coverage is achieved.
- Experiment 3 Overhead: The obfuscated binaries produced by Loki have an overhead factor of 300 to 500 (runtime) and 20 to 50 (size)
- Experiment 04 Key Encodings: The SMT solver cannot solve the Factorization-based key encoding and 70% of the point functions
- Experiment 05 Key Encodings on Binary Level: The SMT solver finds a correct key in 31% of the cases
- Experiment 06 Taint Analysis: Taint analysis taints all but 17% of the instructions
- Experiment 07 Backward Slicing: Backward slicing slices all but 5% to 8% of the instructions
- Experiment 08 Symbolic Execution: SE simplifies no handler (static scenario) or 18% (dynamic attacker; depth 3) / 15% (dynamic attacker; depth 5)
- Experiment 09 MBA Diversity: Loki uses $5,482/7,000 = 78\%$ unique MBAs
- Experiment 10 MBA Formula Deobfuscation: LokiAttack significantly outperforms MBA Blast (the best competitor); resuls should be similar to Figure 3
- Experiment 11 Complexity of Core Semantics: Using superoperators increases the number of core semantics from 16 to 59; with superoperators, the semantic depth ranges from 5 to 13 with a peak at depth 9
- Experiment 12 Limits of Program Synthesis: Synthesis falls of w.r.t. synthesizing expressions of higher semantic depth; shape should be similar to Figure 5
- Experiment 13 Superoperators on the binary level: Syntia manages to synthesize about 19% of Loki's expressions

Due to non-determinism (in both our obfuscator and analysis tooling) and the scope of this artifact evaluation (evaluating 10 binaries instead of $1,000$), we expect quite some fluctuations. In some cases, it may be necessary to evaluate 100 binaries instead of 10 to reproduce our results.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

The artifact evaluation consists of two parts: (1) evaluation of attacks on signed OpenDocument Format (ODF) documents and (2) evaluation of Document Signature Validator (DocSV).

To evaluate the attacks on signed ODF documents, we provide multiple proof of concept (PoC) files. By opening the PoC files the reviewer can evaluate the success of the attack. This success is either code execution via signed macros, content spoofing, or timestamp manipulation. To evaluate all attack classes, at least one Windows 10 system with the affected ODF applications is required. Optionally, a macOS, Linux, iOS, and an Android system is required to evaluate the artifacts under all analyzed ODF applications.

The second part of the artifact evaluation focuses on DocSV – our tool is capable to evaluate the signature status of signed documents in ODF, Office Open XML (OOXML), and Portable Document Format (PDF) formats. We provide a collection of test documents that can be used by the reviewers. Both the source code and the compiled executable are provided for this purpose. For the evaluation of the DocSV tool, a Windows 10 system with the respective ODF, OOXML and PDF applications to be tested is required.

A.2 Artifact check-list (meta-information)

- ☑ Run-time environment
  - Required: Windows 10
  - Optional: macOS Catalina, Ubuntu 20.04.3 LTS, iOS 15, and Android 10
- ☑ Software Installation
  - ODF office applications (see Section A.3.1)
  - DocSV executable (see Section A.3.1)
- ☑ Resources for the Evaluation
  - How much disk space required – approx. 50 GB
  - How much time is needed to prepare workflow – approx. 2h
  - How much time is needed to complete experiments – approx. 1h
- ☑ Code licenses (if publicly available)?: trial licenses are sufficient

A.3 Description

A.3.1 How to access

Access to the Vulnerable Applications Below we have listed links to the installation files of various ODF applications. Please note that not all ODF applications are freely available in the vulnerable version.

- Apache OpenOffice: choose version 4.1.8 here.
- IBM Lotus Symphony 3.0.1: here, fp2-Update: here.
- LibreOffice 7.0.4.2: here.
- Microsoft Office 2019: Microsoft Office 2019 can be downloaded as a trial version here.
- Collabora Online (CODE) 6.0-18: Virtual machine (VM) images for the online variant of Collabora are available here.

Artifacts Stable URL to all artifacts: `https://github.com/RUB-NDS/DocumentSignatureValidator/releases/tag/Artifact_Evaluation`

- PoC files for all attacks described in the paper are available here.
- DocSV source code and a compiled executable can be downloaded on GitHub.
- A collection with test files for DocSV is available here.

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

To evaluate all attack classes, at least a Windows 10 system with the vulnerable ODF applications is required. Optionally, a macOS Catalina, Ubuntu 20.04.3 LTS, iOS 15 and Android 10 system is required to be able to evaluate the artifacts under all analyzed ODF applications.

Due to their same code base, LibreOffice and Collabora Office cannot be installed on Windows at the same time.

For OpenOffice on macOS, the Mozilla Certificate Store must be associated to properly validate the trusted entity certificate (see configuration tutorial).

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

ODF Signature Attacks All ODF applications (see Section A.3.1) need to be installed on a Windows 10 VM or on a physical machine. Optionally, the same for macOS Catalina, Ubuntu 20.04.3 LTS, iOS 15 and Android 10. To evaluate the PoC files on Collabora Online, run the prepared VM on VMware or VirtualBox and follow the instructions.

DocSV To evaluate DocSV, no installation is required, just run `DocumentSignatureValidator.exe`. In addition to ODF applications, DocSV can also be evaluated with PDF applications (Adobe Acrobat Reader DC and Foxit PDF Reader), as well as with OOXML applications (Microsoft Office).

Installation steps of Collabora Online (CODE) VM

1. Import the VM Image.
2. Change the network adapter to NAT in your virtualization software and add a port forwarding rule to forward port 443 to the VM.
3. Start the VM.
4. Enter a City.
5. Use the IP provided by DHCP or one of your own.
6. Select `Manage users and permissions directly on the system`.
7. Enter `test` as organization and add a mail address.
8. Choose a password.
9. Set `code.test.local` as FQDN.
10. Add `code.test.local` with IP `127.0.0.1` to the `etc/hosts` file of the operating system.
11. After the setup is finished, open `https://code.test.local` via the browser and register for a free license.
12. After that you can upload the ODF documents via `https://code.test.local/nextcloud/login`. Login: User name `Administrator`, Password: which was assigned during the installation.

A.5 Experiment workflow

Ground Truth First, the reviewers can see different documents that are: not signed, signed with a trusted key, signed with an untrusted key, and manipulated by invalidating the signature.

Configuration We need to trust the certificate of the `trusted.person.odf@gmail.com` and allow the execution of macros for this person. For this purpose, open the file `ODF_macro_signature_valid_and_trusted.odt`. A dialog then opens asking whether the creator `trusted.person.odf@gmail.com` should be trusted. To do this, check the "Always trust macros from this source" box and then press the "Enable Macros" button.

DocSV: Usage and Configuration DocSV uses XML configuration files for execution. Sample configuration files for evaluating various applications are available here.

- Input Files: In `<files><path></path></files>` users can specify the directory of the documents that will be analyzed. Test documents with different signature statuses and formats are available here.
- Results: The directory for saving the screenshots created during the check, as well as the `CSV` report are specified in `<output><path></path></output>`.
- Detection Rules: In `<sigvalidstring/>`, `<siginvalidstring/>` and `<sigproblem/>` the detection rules for each application can be specified. Note that the default configuration is dependent on the language. To avoid false results, use the English version of the office applications.
- Timeout: DocSV analyzes the application's process memory. To guarantee that the analyzed document is fully loaded into the memory, users can configure a timeout in `<wait/>`. Users with limited PC resources are encouraged to increase the timeout.

DocSV requires one parameter as input – the configuration file: `DocumentSignatureValidator.exe config_file_examples/config_LibreOffice.xml`

Prepare Foxit and Adobe for DocSV Foxit and Adobe use their own certificate stores, so unknown signer certificates must first be set up as trusted.

Adobe:

1. Open PDF test file containing a valid signature.pdf with Adobe Acrobat.
2. Open the Signature Panel.
3. Click on the arrow of the `Rev. 1...` signature.
4. Open `Signature Details→Certificate Details...`
5. Click on `Trust→Add to Trusted Certificates...` to trust the certificate.

Foxit:

1. Open PDF test file containing a valid signature.pdf with Foxit.
2. Open the Siganture Panel (left side `Manage digital signatures`).
3. Right click on the signature `Rev. 1...→Show Signature Properties`.
4. Click on `Show Certificate...`
5. Click on `Trust→Add to Trusted Certificates` to trust the certificate.

DocSV must be added to Foxit as a trusted app. Click on `File→Preferences→Trust Manager→Open Foxit PDF Reader from applications without valid digital signatures→Change Settings...` add the path to the DocSV .exe and click `Allow`.

A.6 Evaluation and expected results

A.6.1 ODF Signature Attacks

All five attacks described in the paper can be evaluated. The PoC files are sorted by the vulnerable ODF applications for this purpose. For each attack class, two ODF documents are included, as well as two folders. Files starting with `01_` represent the initial document. Files starting with `02_` represent the documents manipulated by the attacker. The corresponding directories contain the unzipped ODF file.

01: Macro Manipulation with Certificate Doubling The attacker signs the document with its own key and thus can choose the macro code. The public key of a trusted entity (e.g., `trusted.person.odf@gmail.com`) is included to mask the document as trustful.

- **Execution**: Open the document `02_doc_macros_signed_by_attacker_manipulated.odt`.
- **Expected Result**: After opening the document, macro code is automatically executed which opens a simple message box. In `Tools→Macros→Digital Signatures...`, the trusted entity `trusted.person.odf@gmail.com` is displayed to the victim as the signer, even though the signature was created by the attacker.

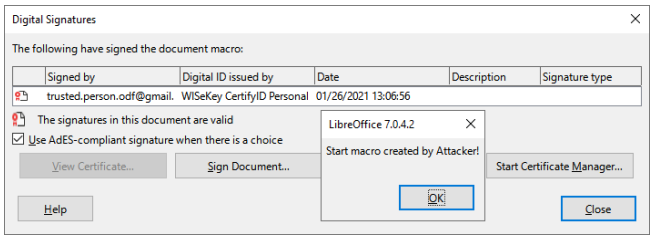


Figure 1: Expected result for `01: Macro Manipulation with Certificate Doubling`

02: Content Manipulation with Certificate Doubling The attacker signs the document with its own key and thus can choose the content of the document. The public key of a trusted entity (e.g., `trusted.person.odf@gmail.com`) is included to mask the document as trustful.

- **Execution**: Open the document `02_doc_signed_by_attacker_manipulated.odt`.
- **Expected Result**: After opening the document, a valid and trusted document signature is displayed to the victim. Under `File→Digital Signatures→Digital Signatures...`, the trusted entity `trusted.person.odf@gmail.com` is displayed to the victim as the signer, even though the signature was created by the attacker.

03: Content Manipulation with Certificate Validation Bypass The attacker signs the document with its own key and thus can choose the content of the document. The attacker disables the verification of the certificate chain and successfully masks the document as trustful.

- **Execution**: Open the document `02_doc_signed_by_attacker_manipulated.odt`.

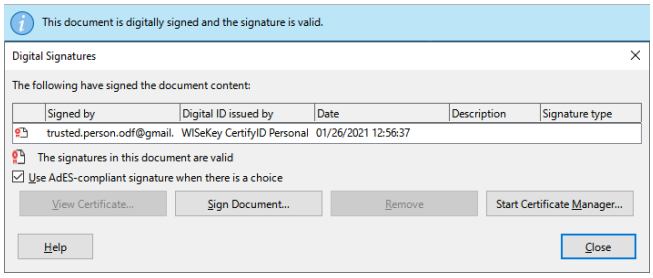


Figure 2: Expected result for `02: Content Manipulation with Certificate Doubling`

- **Expected Result**: After opening the document, a valid and trusted document signature is displayed to the victim. Under `File→Digital Signatures→Digital Signatures...`, a trusted entity arbitrarily chosen by the attacker is displayed to the victim as the signer, even though the signature was created by the attacker.

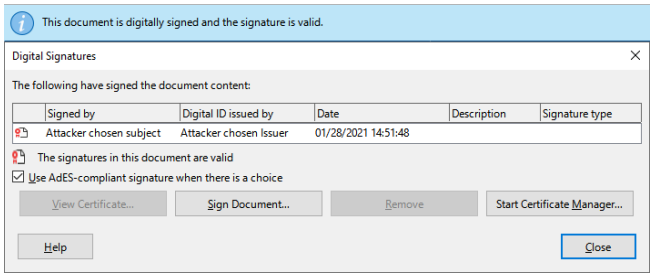


Figure 3: Expected result for `03: Content Manipulation with Certificate Validation Bypass`

04: Content Manipulation with Signature Upgrade The attacker possess an ODF with signed macros that is created by trusted entity. The attacker abuses the partial coverage of the digital signatures and manipulates the content of the document directly due to the missing integrity protection. Thus, the attacker can choose the content of the document arbitrarily.

- **Execution**: Open the document `02_doc_macros_signed_by_trusted_person_manipulated.odt` with Microsoft Office 2019.
- **Expected Result**: After opening the document, a valid and trusted document signature is displayed to the victim. Under `File→View Signatures`, the trusted entity `trusted.person.odf@gmail.com` is displayed to the victim as the signer.

05: Timestamp Manipulation with Signature Wrapping The attacker possess an ODF with signed content that is created by trusted entity. The attacker applies an XML Signature

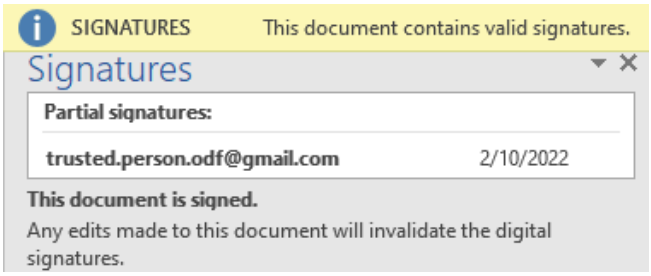


Figure 4: Expected result for `04: Content Manipulation with Signature Upgrade`

Wrapping attack. As a result the attacker can choose any timestamp for the signature.

- ☑ **Execution**: Open the document `02_doc_signed_by_trusted_person_manipulated.odt`.
- ☑ **Expected Result**: After opening the document, a valid and trusted document signature is displayed to the victim. Under `File→Digital Signatures→Digital Signatures...`, the trusted entity `trusted.person.odf@gmail.com` is displayed to the victim as the signer with the attacker's chosen timestamp `66/66/6666 00:00:00`.

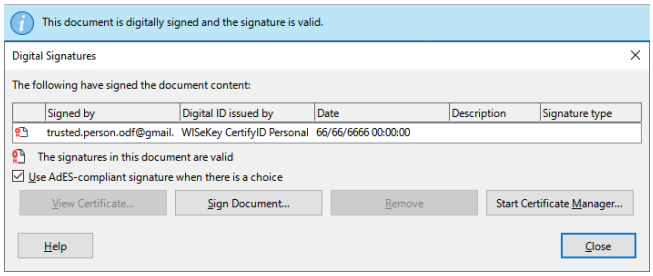


Figure 5: Expected result for `05: Timestamp Manipulation with Signature Wrapping`

Further Macro Exploit Examples If the reviewers are interested in more powerful exploits, we created additional examples. These PoC files contain two more variants of the attack class `01` and are specially designed for Windows.

First variant `exe_download_execute`: The included macro downloads an `.exe` file from https://github.com/attodf/odf-test when the document is opened and saves it to `C:\Users\%USERNAME%\AppData\Local\Temp`, then automatically executes the program. The program is harmless and does not contain any malicious code. It just outputs a text on the console. A working Internet connection is required for this variant.

Second variant `ransomware`: The included macro creates the file `example_ransomware.py` under `C:\Users\%USERNAME%\AppData\Local\Temp`. Then, this Python script is executed, using the Python environment of the respective office application, which can be found under `C:\Program Files\%ODF-Application%\program\python.exe`. This ransomware simulation serves as a PoC and is not supposed to do any damage, so it only creates a hashed file with `.hashed` extension from each file under `C:\Users\%USERNAME%\Desktop`. The function to delete the original files is not active in the Python code.

A.6.2 DocSV

DocSV can be used to check the signature status of signed documents in ODF, PDF and OOXML formats. DocSV is started by `DocumentSignatureValidator.exe` via the console. The configuration is done using an XML configuration file which must be passed as argument (see Section A.5). DocSV automatically opens the individual signed documents and determines the signature status through a memory analysis. The analysis results are exported as a `CSV` file. In addition, a screenshot of the opened document is also saved.

- ☑ **Execution:** Start DocSV and pass one of the prepared configuration files. DocSV will test the files stored in the folder test_documents. As part of the artifact evaluation, you can generate your own files and test these with DocSV.
- ☑ **Expected Result:** DocSV produces a report containing the result of the analysis and saves this report together with the taken screenshots in results_dovsv. Our collection of test documents contains one unsigned, signed, and manipulated document. DocSV should detect these correctly.

A.7 Experiment customization

N/A

A.8 Notes

The certificate of the trusted entity is valid until 11st May 2022. For the evaluation of the attacks after this date, it is necessary to reset the date of the operating system accordingly.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

ARTIFACT EVALUATED
usenix ASSOCIATION
FUNCTIONAL

A Artifact Appendix

A.1 Abstract

Obligatory. Briefly describe your artifact including minimal hardware and software requirements, how it supports your paper, how it can be validated, and what is the expected result. At submission time, it will also be used to select appropriate reviewers. It will also help readers understand what was evaluated and how.

The artifact is the source code of the tool (i.e., PaymentScope) we proposed in the paper. It can detect payment bypass vulnerability in Unity mobile games. It is implemented atop **Ghidra**. To evaluate PaymentScope, we have attached 15 games in which 10 of them are vulnerable. PaymentScope can detect that 10 of them are vulnerable and it can tell the vulnerability type (i.e., local-verification or no-verification). To run PaymentScope, we have prepared a VirtualBox VM in which all the requirements have been setup. The VM needs 2 cores CPU and 8GB memory (mostly required by Ghidra)

A.2 Artifact check-list (meta-information)

Obligatory. Fill in whatever is applicable with some keywords and remove unrelated items.

- **Program:** the source code of PaymentScope
- **Run-time environment:** VirtualBox VM with 2 cores CPU and 8GB memory
- **Security, privacy, and ethical concerns:** please don't use the tool to attack any real games.
- **Output:** identify 5 local-verification and 5 no-verification games
- **Experiments:** run `/home/paymentscope/Desktop/runPaymentScopeOnTestData.py` in the VM
- **How much disk space required (approximately)?:** 30GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour depending on the Internet bandwidth
- **How much time is needed to complete experiments (approximately)?:** less than 1 hour

A.3 Description

Obligatory. For inapplicable subsections (e.g., the "How to access" subsection when not applying for the "Artifacts Available" badge), please specify 'N/A'.

A.3.1 How to access

N/A

A.3.2 Hardware dependencies

2 cores CPU and 8GB memory

A.3.3 Software dependencies

VirtualBox

A.3.4 Data sets

15 games. Among them, 10 are vulnerable. In particular, 5 are local-verification and 5 are no-verification games.

A.3.5 Security, privacy, and ethical concerns

Please don't use the tool to attack any real games.

A.4 Installation

Obligatory. Describe the setup procedures for your artifact targeting novice users (even if you use a VM image or access to a remote machine).

- Read the `README.md` file for the source code in `Source Code` folder
- Read the `README.md` file for the Virtual Machine in `Virtual Machine` folder
- Install the VirtualBox

A.5 Experiment workflow

Describe the high-level view of your experimental workflow and how it is implemented, invoked and customized (if needed), i.e. some OS scripts, IPython/Jupyter notebook, portable CK workflow, etc. This subsection is optional as long as the experiment workflow can be easily embedded in the next subsection.

- Login to VM
- Run `/home/paymentscope/Desktop/runPaymentScopeOnTestData.py` to conduct the experiments
- For each output folder, find the `isVulnerable` field in `analysisRes.json` file. The field indicates whether the game is vulnerable and the vulnerability type.

A.6 Evaluation and expected results

Obligatory. Start by listing the main claims in your paper. Next, list your key results and detail how they each support the main claims. Finally, detail all the steps to reproduce each of the key results in your paper by running the artifacts. Describe the expected results and the maximum variation of empirical results (particularly important for performance numbers).

Main claim: PaymentScope can detect payment bypass vulnerability in Unity mobile games. It is implemented by the guidance of the Algorithm 1 in the paper.

To support the claim, we have attached a PDF in the source code to map our implementation to the Algorithm 1. In addition, we have attached 15 games for testing, in which 5 are local-verification and 5 are no-verification games. We have manually verified the 10 games and they are indeed vulnerable. The games and the vulnerability types is explained in file `PaymenScope/Virtual Machine/README.md`.

After run `runPaymentScopeOnTestData.py`, the vulnerability type can be found in `isVulnerable` field in `analysisRes.json` file which is located in the output folder. 5 of them should be no-verification, 5 of them should be local-verification and the rest should be 'secure'.

A.7 Experiment customization

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

The public repository[1] contains all the code necessary to reproduce the data for all performance graphs/tables in the paper, as well as PoCs to demonstrate that the mitigation works. This includes patches and build instructions for LLVM11, the Intel SGX SDK and PSW, as well as the benchmarks. The artifact requires SGX to evaluate, and is easiest to run on Ubuntu 18.04 or 20.04.

A.2 Artifact check-list (meta-information)

- **Program:** Adapted versions of `nbench` and `sgxbench` are downloaded & installed via included scripts.
- **Compilation:** Requires a modified Clang 11, install & download script is included.
- **Transformations:** A tool to fix up relocations is included (relocator).
- **Run-time environment:** Needs a native Linux installation that supports SGX, Ubuntu 18.04 or 20.04 are strongly recommended. Build scripts need internet access at several points. Requires root for installation and evaluation. PoCs require the PTEditor kernel module.
- **Hardware:** Intel CPU with SGX support, needs to be vulnerable to LVI-Null for PoC tests (affected CPUs).

  The PoCs need a kernel module, which means either self-signing or disabling secure boot. This may require physical access to the machine.
- **Run-time state:** As this artifact includes performance benchmarks, a stable CPU frequency and isolated cores are recommended.
- **Execution:** For ideal testing, the system should have isolated cores, fixed frequency, and not much other activity.
- **Metrics:** Benchmarks report cycle count or iterations/s, PoCs report leakage percentage.
- **Output:** Benchmark outputs are .csv tables with performance, an included spreadsheet can convert to a graph similar to the paper.
- **Experiments:** Installation scripts are included and described here and in READMEs.
- **How much disk space required (approximately)?:** 4-5GB
- **How much time is needed to prepare workflow (approximately)?:** 2-3h
- **How much time is needed to complete experiments (approximately)?:** 3-6h, depends on hardware
- **Publicly available?:** `https://github.com/IAIK/LVI-NULLify`
- **Code licenses (if publicly available)?:** zlib

[1] `https://github.com/IAIK/LVI-NULLify`

A.3 Description

A.3.1 How to access

Clone `https://github.com/IAIK/LVI-NULLify/tree/ae_final` and follow the README.md from there.

A.3.2 Hardware dependencies

As this is a mitigation for Intel SGX, SGX support is a hard requirement. To fully evaluate the PoCs, and not just mitigation performance, the CPU also needs to be vulnerable to LVI. You can check if your CPU is vulnerable here: `https://software.intel.com/content/www/us/en/develop/topics/software-security-guidance/processors-affected-consolidated-product-cpu-model.html`

A.3.3 Software dependencies

We strongly recommend Ubuntu 18.04 or 20.04 as these are officialy supported by Intel, and all our tools were tested on them.

Beyond standard compilation tools (ninja, cmake etc) our PoCs require the PTEditor kernel module [2]. Other requirements are listed in the README files at the appropriate points.

A.4 Installation

Follow the detailed README in the top-level directory to set up our modified clang compiler and relocator and install the SGX driver as well as our modified SGX SDK and PSW.

Once that is done, you can already test your installation with the PoCs by following the README file in the POC directory.

With a working PSW and driver, you can follow the README in the benchmarks directory to download and build the benchmarks.

A.5 Experiment workflow

After building the benchmarks, follow along in the README to start all or a subset of them. An important aspect to keeping benchmarks comparable is to fix the CPU's frequency to a sustainable level, and idealy run them on an isolated core.

PoCs can be run according to the README in the POC folder.

A.6 Evaluation and expected results

The main results in our paper are contained in Figure 4/Table 3. These are the performance overheads of our LVI-NUll mitigation compared to other, similar mitigations. The second, more implicit result is the efficacy of LVI-NULLify.

For the benchmarks, the absolute performance overheads vary significantly between different machines and architectures (compare Figure 4 and Figure 5), but the relative differences should be roughly similar. That is: LVI-Nullify should be the fastest mitigation, or at least very close to Intel CFI, typically followed, with some distance, by Intel's optimized-cut mitigation.

For the PoCs, starting once without and once with mitigation should produce qualitatively similar results to the examples shown in the README. That means, for the 3 PoCs where LVI-Nullify is

[2] `https://github.com/misc0110/PTEditor/`

effective, leakage rate should drop to zero, or a level that is comparable to the noise-catching output "other". While absolute leakage rates before applying the mitigation may differ significantly from system to system, they should be clearly differentiable from "other".

The respective READMEs for benchmarks and PoCs detail how to reproduce these results.

A.7 Experiment customization

Attack PoCs need a cache miss threshold, which is automatically determined. If this doesn't work, it can be set manually in the corresponding *App.cpp* file. All PoCs include a conf.h file, in which the character that should be leaked can be changed if desired.

Both benchmark run-scripts contain a variable called "isolated_core" that sets the core on which they should be run on. Set this to an isolated core, if available.

sgx-nbench contains a parameter to change the number of iterations in the file, see the benchmarking README.





A Artifact Appendix

A.1 Abstract

This document demonstrates the artifact evaluation of LIGHTENCLAVE, which uses MPK to provide intra-enclave isolation within SGX enclaves. We incorporate LIGHTENCLAVE into two SGX libOSes (Graphene-SGX and Occlum) and carry out evaluations to show the performance of Graphene-SGX/Occlum with and without LIGHTENCLAVE. We provide a remote machine as the SGX, PKU (MPK), and MPX CPU features are needed. According to the specification, AE reviewers can firstly build the tested applications and libOSes, and then carry out all experiments mentioned in paper. The experiments will reproduce the results and generate figures and tables in the paper.

We do not apply for the Available badge because one of the founders currently does not allow to open source the work.

A.2 Artifact check-list (meta-information)

- **Program:** The libOSes used in the experiment are Occlum (commit 0a06c898) and Graphene-SGX (commit 9c226c9a). The applications and libraries used in the experiments are SGX-OpenSSL (commit 5bacfaf), SGX-SQLite3 engine (v3.23.0), Lighttpd (v1.4.40), GCC (v4.4.5), Fish shell (v3.0.0) and BusyBox (v.1.23.1)
- **Hardware:** An Intel x86 platform that supports Intel SGX, PKU and MPX.
- **Run-time environment:** The experiment is carried out on Linux. The kernel should set *CR4.FSGSBASE = 1* to allow userspace applications use *wrfsbase* and *wrgsbase* to modify fs.base and gs.base. On newer versions of Linux (>= 5.9), *CR4.FSGSBASE = 1* is always set. The SGX SDK 2.4 and SGX Driver 2.4 should be installed on the host. Docker is used as the building environment. We also use the runtime environment provided by Occlum and Graphene-SGX.
- **Metrics:** We use the applications' throughput and execution latency of operations to study LIGHTENCLAVE's performance.
- **Output:** Some of the outputs are numerical results that can be compared with tables in the paper. The other outputs are figures that are available in the paper.
- **Experiments:** We provide scripts that reproduce the experiment results and generate figures and tables in the paper.
- **How much disk space required (approximately):** Around 18G. We suggest different AE reviewers use different working directories. Since the disk space of our remote machine is limited, please remove the working directory once the artifact evaluation completes (in case leading to out-of-disk for others).
- **How much time is needed to prepare workflow (approximately):** We provide a remote machine which is setted up for artifact evaluation so that reviewers do not need to prepare the workflow.
- **How much time is needed to complete experiments (approximately):** The building procedure is about 1 hour. The complete evaluation takes about 3 hours.

A.3 Description

A.3.1 How to access

N/A

A.3.2 Hardware dependencies

The artifact evaluation requires an Intel x86 platform that supports Intel SGX, PKU and MPX. Our remote machine has an Intel i7-10700 IceLake CPU.

A.3.3 Software dependencies

Except for the environment mentioned in the checklist, LIGHTENCLAVE requires the building system and toolchain from Occlum and Graphene-SGX. To save the time for reviewers, we have prepared the software dependencies for the artifact evaluation in the offered machine.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

The installation procedure requires building the libOSes (Graphene and Occlum with and without LIGHTENCLAVE) and the compilation toolchain. After that, we compile the applications for the evaluation. We provide a remote machine with software dependencies prepared, where reviewers can start from the installation stage. Please check *Artifact access* in the submission (on hotcrp) for detail.

After login into the machine, the home directory contains the *lightenclave-artifact* directory that holds evaluation materials. Before starting the evaluation, please copy *lightenclave-artifact* to another directory to avoid conflicts between different reviewers.

```
> sudo cp -r lightenclave-artifact your-
    evaluation-directory
> cd your-evaluation-directory
```

We firstly build Occlum toolchain and applications running inside Occlum. We use docker as the building environment.

```
> bash build_occlum_apps_in_docker.sh
# Now inside the docker
# Takes long time: about 45 minutes
> bash occlum/build_toolchain_and_app.sh
> exit
```

Then we build applications running inside Graphene using another docker environment. This docker environment is used in the

following building and evaluation workflow, including building Occlum and Graphene-SGX libraries and applications running without libOSes (for Figure 8).

```
> bash reproduce_in_docker.sh
# Now inside the docker
> bash graphene-sgx/build_app.sh
> bash occlum/build_libos.sh
> bash graphene-sgx/libos/build_libos.sh
> bash sdk-bench/prepare.sh
```

A.5 Experiment workflow

N/A

A.6 Evaluation and expected results

We claim that: 1. LIGHTENCLAVE is fast in terms of light-enclave creation and communication. 2. LIGHTENCLAVE incurs low performance overhead for intra-enclave isolation to applications. 3. LIGHTENCLAVE improves the performance in real-world scenarios in existing LibOSes.

For the 1st claim, Table 2 shows the task creation latency in SGX libOSes. When incorporated with LIGHTENCLAVE, the application creation time is shortened. The results can be reproduced by executing:

```
# about 8 minutes
> ./scripts/table2.py
```

Then Figure 7 demonstrates that LIGHTENCLAVE can provide fast enclave communication using shared memory between light-enclaves. In contrast, the communication in Graphene is more time-consuming due to data encryption. The figure can be reproduced by executing:

```
# Takes about 50 minutes
> ./scripts/figure7.py
> gnuplot -p ./plots/figure7.plt
# The figures locate at plots/figure7a.eps
    and plots/figure7b.eps
```

For the 2nd claim, we use LIGHTENCLAVE to isolate sensitive code from third-party code for security. We compare it with Nested Enclave, which uses an inner enclave to isolate third-party code. Figure 8a isolates OpenSSL library from the application. Figure 8b isolates SQLite3 library from a key-value store server. The figure can be reproduced by executing:

```
# Takes about 7 minutes
> ./scripts/figure8a.py
# Takes about 3 minutes
> ./scripts/figure8b.py
> gnuplot -p ./plots/figure8a.plt
# The figure locates at plots/figure8a.eps
> gnuplot -p ./plots/figure8b.plt
# The figure locates at plots/figure8b.eps
```

For the 3rd claim, we apply LIGHTENCLAVE to Occlum and Graphene and test real-world applications' performance. The applications are Lighttpd, GCC, Fish Shell and some serverless functions.

We configure Lighttpd with two isolated workers and use ApacheBench to get the throughput. LIGHTENCLAVE improves performance in Occlum since there is no boundary checking. Figure 9 shows the results, which can be reproduced by executing:

```
# Takes about 4 minutes
> ./scripts/figure9.py
> gnuplot -p ./plots/figure9.plt
# The figure locates at plots/figure9.eps
```

The fast task creation in LIGHTENCLAVE benefits GCC, which frequently forks processes for compilation. We isolate each GCC-related processes (*cc1*, *as*, *collect2* and *ld*) in the enclave. And we compile five applications with various sizes of code bases. Figure 10 shows the results, which can be reproduced by executing:

```
# Takes about 16 minutes
> ./scripts/figure10.py
> gnuplot -p ./plots/figure10.plt
# The figures locate at plots/figure10a.eps
    and plots/figure10b.eps
```

We then evaluate Fish Shell's performance by invoking several BusyBox commands (od, sort, grep, wc etc.) for text processing. LIGHTENCLAVE improve the performance by creating tasks fast (compared with Graphene) and avoiding SFI overhead (Compared with Occlum). Table 3 shows the result, which is reproduced by:

```
# Takes about 8 minutes
> ./scripts/table3.py
```

The fast task creation in LIGHTENCLAVE reduces initialization overhead in FaaS scenarios. We evaluate four serverless functions' execution latency to show the benefits. LIGHTENCLAVE is compared with initializing a new enclave before execution (COLD) and using an existing enclave for execution (WARM). In theory, LIGHTENCLAVE and WARM have similar execution latency while it takes fewer resources. Figure 11 shows the results, which can be reproduced by:

```
# Takes about 40 minutes
> ./scripts/figure11.py
> gnuplot -p ./plots/figure11.plt
# The figure locates at plots/figure11.eps
```

A.7 Experiment customization

N/A

A.8 Notes

If the experiments freeze (the execution time is far beyond the time we offer), reviewers can kill the docker and restart the experiment. Maybe the libOSes with the specified commits contain some unknown issues.

```
# Press Ctrl+C. Or use docker kill command
> exit
# Re-enter the docker
> bash reproduce_in_docker.sh
# e.g., ./scripts/figure10.py fails
```

```
> ./scripts/figure10.py
```

After the artifact evaluation, please remove the working directory as it consumes large disk space.

```
> sudo rm -rf your-evaluation-directory
```

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

SGXFuzz presents a novel approach to fuzz SGX enclaves in a user-space environment including the synthesis of ECall structures that automatically synthesizes a nested input structure as expected by the enclaves using a binary-only approach. The prototype consists of an enclave dumper that extracts enclaves memory from distribution formats, a fuzzing setup to fuzz extracted enclave, as well as a series of scripts to perform result aggregation. The fuzzing setup is the core of SGXFuzz and is built upon the kAFL fuzzer and the Nyx snapshotting engine. We extend the existing code of kAFL to accommodate our structure synthesis in Python. The Nyx fuzzing engine utilizes the Intel PT CPU extension to get code coverage information but does not contain any changes for SGXFuzz. Finally, we provide several scripts to process the crashes found during the fuzzing campaigns as well as the synthesized structure layouts.

A.2 Artifact check-list (meta-information)

- **Compilation:**
  recent cmake, gcc/g++, c++20, Ubuntu 22 recommended
- **Transformations:**
  custom binary-to-binary included
- **Binary:**
  Ubuntu 5.10.75 kernel
- **Run-time environment:**
  Linux/Ubuntu, custom kernel included, root access, bare metal/no VM
- **Hardware:**
  Intel CPU, Skylake or newer (Intel-PT-capable)
- **Metrics:**
  Structure Layouts, Crashes/Vulnerabilities/Bugs, Coverage
- **Output:**
  Terminal, Files (msgpack/structures, edges, crashing payloads)
- **Experiments:**
  - Extract enclaves
  - Run the Fuzzer (compile runner, start fuzzing)
  - Post-process/aggregate results
- **How much disk space required (approximately)?**
  3.5 GB install size + temporary 10–30 GB
- **How much time is needed to prepare workflow (approximately)?**
  3 h
- **How much time is needed to complete experiments (approximately)?**
  Full experiment:
  24 h per run, 30 main evaluation runs, 80 ablation runs
  = 110 days using a single machine (+ data aggregation)
  Minimal sample evaluation: 1 h
- **Publicly available (explicitly provide evolving version reference)?**
  https://github.com/uni-due-syssec/sgxfuzz/
- **Code licenses (if publicly available):**
  MIT, BSD, GPL, AGPL, Apache (see individual components)
- **Workflow frameworks used?**
  Bash and Python
- **Archived (explicitly provide DOI or stable reference):**
  https://github.com/uni-due-syssec/sgxfuzz/tree/usenix2022

A.3 Description

We will now describe the components of the artifact, how they are related and what each component is used for. The SGXFuzz artifact consists of the enclave dumper, enclave runner, the fuzzing setup, and the enclaves evaluated in the paper. The enclave dumper extracts the enclave memory from enclave distribution formats (cf. Section 5.1). This step has to be done only once per enclave, and we have already performed that step for all enclaves. The enclave runner uses the previously extracted enclave memory to run the enclave as a regular user-space process (cf. Section 5.2). The runner is a C++ program that loads the enclave memory, handles the emulation of the context switch that would usually be performed by the SGX instruction set and performs the structure allocation for each input. Finally, our fuzzing setup consists of a front end that generates fuzzing inputs and performs the structure synthesis, and a back end that executes the target and collects coverage. We use kAFL as a foundation for our fuzzing front end and add new code to the fuzzer to perform the structure synthesis. The back end consists of a patched version of QEMU and KVM to allow the collection of coverage data using the Intel PT CPU extension. We did not perform any modifications on the fuzzing back end.

A.3.1 How to access

All code relevant to the artifact and links to the components required for the fuzzing setup are publicly available on GitHub https://github.com/uni-due-syssec/sgxfuzz/tree/usenix2022.

A.3.2 Hardware dependencies

Our fuzzing back end consisting of a modified QEMU and KVM uses the Intel PT CPU extension to collect coverage data. Thus, an Intel PT-enabled CPU is required to use our fuzzing setup. However, Intel PT does not work in a virtualized environment and as such, cannot run in VM. Notice that the Intel SGX is not required at any point.

A.3.3 Software dependencies

Generally, any Linux distribution should be able to run our artifacts. However, we only tested it on Ubuntu 22.04 and the scripts we provided to set up the fuzzing setup were developed and tested with Ubuntu 22.04 in mind.

A.4 Installation

We include a setup script that should perform the major steps.

First, disable SGX in the BIOS if supported by the CPU.

Clone the repository.

Install required packages:

```
sudo apt install \
  python2 python3 libpixman-1-dev pax-utils bc \
  make cmake gcc g++ pkg-config unzip \
  python3-virtualenv python2-dev python3-dev \
  libglib2.0-dev
```

Then, you can use setup.sh to compile and install the components, or follow the steps manually. That is:

- Initialize the submodules:
  `git submodule update --init --recursive --depth=1`
- QEMU-Nyx:
  https://github.com/nyx-fuzz/QEMU-Nyx#build
- KVM-Nyx:
  https://github.com/nyx-fuzz/KVM-Nyx#setup-kvm-nyx-binaries
- (Virtual) environments for python2 and python3 and install
  - python2: configparser mmh3 lz4 psutil ipdb msgpack inotify
  - python3: six python-dateutil msgpack mmh3 lz4 psutil fastrand inotify pgrep tqdm hexdump
- Install zydis (`cd zydis && cmake . && make install`)

A.5 Experiment workflow

The experiment workflow includes three main parts: Enclave dumping, Fuzzing, Result aggregation.

A.5.1 Enclave Dumping

First, enclave dumping is used to extract the enclave memory. It is based on the Linux SGX SDK. By providing the enclave dumper with `enclave.signed.so`, a memory dump with the name `enclave.signed.so.mem`, a memory layout `enclave.signed.so.layout`, and the address of the enclave's entry point (specifically the offset of the TCS) `enclave.signed.so.tcs.txt`.

Compile is using:

```
make -C ./enclave-dumper/
```

Run it using:

```
./enclave-dumper/extract.sh [enclave.signed.so]
```

A.5.2 Fuzzing

To fuzz the previously extracted enclave, several steps are involved. We bundled all of them together in a script that runs a minimal fuzzing test:

```
./run-example.sh
```

The script runs the following steps automatically. First, the enclave runner is compiled using

```
make-enclave-fuzz-target.sh enclave.signed.so.mem \
  enclave.signed.so.tcs.txt
```

The result of the compilation is a `fuzz-generic` binary, which is the user-space version of the enclave, and a `liblibnyx_dummy.so`, which is required for the fuzzer.

In the next step, the fuzzing target is packed into a VM that is executed using the QEMU-KVM setup. The packer script can be called as follows

```
nyx_packer.py <enclave-runner> <fuzz-folder> m64 \
  --legacy --purge --no_pt_auto_conf_b \
  --fast_reload_mode --delayed_init
```

Finally, the fuzzing can be started using the kAFL fuzzing frontend. The exact command can be found in the `run_example.sh` script.

If desired, manually crafted seeds can be added to the `imports` folder. Each seed is a file consisting of the ECall ID, the serialized structure definition, and the contents of the buffers.

A.5.3 Result Aggregation

Display synthesized structures:

```
display-structs.py <path/to/fuzzing-workdir> \
  <ecall_index>
```

The script displays the evolvement of the synthesized structure in a tree format for each ECall index, with the ECall index being zero-based. The leaves show the final evolvement of the synthesized structures. Each leave shows the synthesized structure in a specific format.

Structures are serialized, e.g., `40 2 C8 4 0 C24 7 0`, and read left to right. This string denotes a structure of **40** Bytes, which has two (**2**) child structures (**C**). The first child is at offset **8** of the parent and is defined the same way: A size of **4** and zero (**0**) children. The second child has a size of **7** and also zero children. Further, the sizes may be annotated with their address (`40:0x7ffff7faafd8`). Additional types include buffers partially (on the edge) of the enclave's memory (P) and SizeOf (S) buffers of which the size is written to a defined offset.

This script shows how to parse and dump these strings:
`kafl/kAFL-Fuzzer/fuzzer/technique/struct_recovery.py`

Display crashes:

```
analyze_crashes.py <eval-dir> \
  -0 --np --no-ptr-0x7ff --no-large-diff
```

`analyze_crashes.py` script iterates through all crashes found by the fuzzer. This includes the crashes due to implementation artifacts mentioned in Section 5.5. The script performs the filtering according to Section 5.5 and will only display valid crashes. However, manual duplication of the crashes is required to find the real number of unique bugs. The flags supplied to the script do the filtering according to the described filtering techniques in the paper. The script displays useful information to understand the crash: the ECall ID, the signal (usually Segmentation Fault), pc (absolute/relative), the disassembled instruction, and addresses used for memory access.

Calculate Coverage:
`calculate-coverage.sh <path/to/fuzzing-workdir>`
Note that we recalculated the numbers of basic blocks using the basic block semantic of Binary Ninja to provide numbers comparable to TeeRex.

A.6 Evaluation and expected results & Experiment customization

The main goal of the fuzzing process is to find crashing inputs, i.e. vulnerabilities, and the synthesized structure layouts that the enclave calls expects for its input. To ensure that the artifact is functional, any enclave from the previously provided links with enclaves can be used for fuzzing, i.e., `synaTEEv2-20211105` which is the *Synaptics Fingerprint Driver Enclave* from the paper. Fuzz the enclave using the steps shown in `run_example.sh` for 960 core-hours. To fuzz another target than the example, change the `ENCLAVE_PATH` variable to the target path and change the enclave name `enclave.signed.so` to the target enclave's name, i.e., `synaTEE.signed.dll`.

After that, it is possible to use the previously described workflow to display synthesized structures using the `display-structs.py` script on the fuzzing workdir. To analyze the crashes, the workflow to display the crashes can be used. However, manual reverse engineering is required to deduplicate bugs.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact describes the frameworks used for our evaluations. The artifact consists of two relatively separable components: (1) a covert channel discovery framework that runs directly on real hardware, and (2) a gem5 simulation infrastructure that evaluates our mitigation strategies. The covert channel framework can be used to replicate the results presented in Section 4, and in particular, the results summarized in Table 1. The gem5 infrastructure can be used to replicate the performance and security results presented in Section 7.

A.2 Artifact check-list (meta-information)

- **Programs:** SPEC CPU2017, SunSpider JS Benchmarks, WolfSSL RSA and AES benchmarks.
- **Compilation:** LLVM with -O3 for SPEC.
- **Hardware:** AMD Ryzen Threadripper 3960X and Intel Core i7-6770HQ for Covert Channel Framework. The gem5 simulator runs on any modern x86 hardware.
- **Run-time Environment:** The provided covert channel framework and gem5 simulator are tested on Ubuntu 20.04. The scripts for running SPEC benchmarks on gem5 assume an available Slurm workload manager.
- **Output:** The covert channel framework outputs the covert channels bandwidth and error rate. The gem5 simulator outputs the execution time and performance in terms of Cycles Per Instruction (CPI).
- **Experiments:** Scripts and instructions are provided in the artifact README files.
- **How much disk space required (approximately)?:** About 250 GB of disk space is required for the SPEC 2017 simpoints, and around 5 GB is needed for the gem5 code and binaries. The covert channel framework requires less than 100 MB.
- **How much time is needed to prepare workflow (approximately)?:** About 30 minutes to download the frameworks and install requirements, and around 30 minutes to compile gem5.
- **How much time is needed to complete experiments (approximately)?:** Assuming enough available parallelism, the gem5 experiments need at least 3 hours. The covert channel measurements require about 2 hours to complete.
- **Publicly available?:** Yes, the code is available on Github (see Section A.3.1).
- **Code licenses:** GPL v3.

A.3 Description

A.3.1 How to access

The artifact is available on github at the following URL: https://github.com/mktrm/SecSMT_Artifact/tree/86286e06f6f1d8ce9583af950edacb87f14e39ba.

A.3.2 Hardware dependencies

A bare-metal machine is required to run the covert channel measurements. The bandwidth of the covert channels are measured on two specific processors: Intel Core i7-6770HQ and AMD Ryzen Threadripper 3960X. While the covert channel framework can be adapted for other processors, it requires extensive parameter fine-tuning to achieve the best channel.

A.3.3 Programs

We provide Simpoints created from SPEC 2017 benchmarks for the artifact evaluators, but we cannot publish them as they are under copyright. Other programs used for evaluations are publicly available.

A.4 Installation

The artifact provides scripts to install requirements as well as building the provided tools.

A.5 Experiment workflow

This section provides a high-level overview of the experimental workflow. Please follow the instructions in the README for a detailed, step-by-step guide.

The covert channel framework needs to first install a kernel module that facilitates reading values for performance counters. Note that those performance counter values are only used for debugging purposes and our covert channel communication is entirely based on execution time (cycle time). To make a fair comparison between the covert channels, we make sure the processors are configured to be always on performance mode (scripts are provided). Then, the provided makefile detects the hardware (Intel or AMD) and compiles the covert channel measurement codes for all the available covert channels for that platform. It then runs multiple rounds of the experiments for each channel and reports the average bandwidth and error rate of all successful channels (if the error rate is less than 10%).

For the evaluation of the mitigation strategies, we need to first compile the gem5 code and prepare our benchmark programs. Then, we run multiple gem5 simulations for each pair of benchmark programs. Each experiment is configured to represent one of the following multithreading approaches: (1) a fully dynamically shared insecure baseline, (2) a fully statically partitioned pipeline, (3) our Adaptive partitioning, and (4) our Asymmetric SMT approach in which we apply Asymmetric SMT on top of adaptive partitioning for some resources. Finally, once the simulations are finished, we run the provided scripts to extract the results from gem5 simulations and draw the figures.

A.6 Evaluation and expected results

A successful run of the covert channel framework will result in a set of bandwidth and error rate pairs. If the measurements

take place on the mentioned processors, the bandwidth numbers should be around the results reported in Table 1. Note that the fluctuation in bandwidth and error rate numbers can be caused by various sources of noise such as voltage/frequency scaling, OS scheduling, etc.

The gem5 simulation of the mitigation strategies should result in performance numbers that match those presented in the paper.





A Artifact Appendix

A.1 Abstract

All experiments were conducted in Ubuntu-18.04 with 1TB memory and Intel(R) Xeon(R) Gold 6248 20 Core CPU @ 2.50GHz * 2. But it's ok if we don't have that much computing resource. The minimal configuration is at least 4 core CPU, 8G memory and at least 200G disk space. It's recommend to enable more CPU cores, they will speedup the compiling, fuzzing and symbolic execution significantly.

In our paper, we test more than 1000 bugs and each of them require 3 hours kernel fuzzing, 1 hour static analysis and 4 hours symbolic execution. If we plan to accomplish all 1000 cases, it costs more than two weeks, therefore we only choose a subset of them for Artifact Functional.

A.2 Artifact check-list (meta-information)

- **Program:** SyzScope now open source at https://github.com/seclab-ucr/SyzScope/tree/b1a6e20783ba8c92dd33d508e469bc24eaacaab6. This version is the one we conduct the experiment. It's recommend to download the docker container we provided. The details shows in the github README. If you have a fast internet speed, you may want to pull the `ready2go` docker image, otherwise `mini` docker image requires extra compliation.
- **Compilation:** If you pull the `mini` docker image or run SyzScope in your custom system, you have to compile the essential tools. Using command *python3 syzscope –install-requirements*. The detailed instructions can be found at our github page https://github.com/seclab-ucr/SyzScope/
- **Data set:** Since running all cases takes more than two weeks, we only prepare a subset of them for the artifact functional badge. The dateset we provide is the ones we got CVEs from: https://sites.google.com/view/syzscope/home. We made a google sheet of this dataset at https://docs.google.com/spreadsheets/d/16tt4Mo40iyWeuxOXBpRtmV_Zjdddta9nIy-poEuq66E/edit?usp=sharing
- **Run-time environment:** SyzScope is designed on Ubuntu 18.04, written by python3, C++, golang, and bash script. Every other Linux system should support SyzScope just fine. But we still recommend to use our docker image in case any environment differences.
- **Output:** We wrote detailed tutorial for how to read output from fuzzing, static analysis and symbolic execution. You can access them on
  https://github.com/seclab-ucr/SyzScope/blob/master/tutorial/fuzzing.md
  https://github.com/seclab-ucr/SyzScope/blob/master/tutorial/static_taint_analysis.md
  https://github.com/seclab-ucr/SyzScope/blob/master/tutorial/sym_exec.md
- **Experiments:** To run the experiment, you first need to prepare the case hash for SyzScope. Since we already give the dataset, you just need to simply copy and paste the case hash into a file (one hash per line), let's say the name of that file is *dataset*, and run SyzScope with `nohup python3 syzscope -i dataset -RP -KF -SA -SE -timeout-kernel-fuzzing 3 -timeout-static-analysis 3600 -timeout-symbolic-execution 14400 -guided -be-bully &`, If you have enough CPU cores, you can even try run multiple cases at the same time by specify `-pm`, for example `-pm 8` means run 8 cases at the same time. The log output will be written into nohup.out since we use `nohup` to make the process running in background.
- **How much disk space required (approximately)?:** SyzScope requires 24GB for essential packages and tools. Besides them, SyzScope may require 2GB for each case. Considering 8 cases in our dataset, it's better to have 20GB remaining. Therefore in total we suggest having 50GB reamining on your disk.
- **How much time is needed to complete experiments (approximately)?:** At maximum each case takes 3 hours kernel fuzzing, 1 hours static analysis and 4 hours symbolic execution (8 hours in total), but some cases may terminate early. If we run these 8 cases together `-pm 8`, we should finish all of them in 8 hours, if we run them one by one, it probably takes more than 3 days (64 hours).
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License
- **Data licenses (if publicly available)?:** MIT License
- **Archived (provide DOI)?:** https://github.com/seclab-ucr/SyzScope/tree/b1a6e20783ba8c92dd33d508e469bc24eaacaab6 is the stable version that conduct all experiments.

A.3 Description

A.3.1 How to access

Access the github repo at https://github.com/seclab-ucr/SyzScope/tree/b1a6e20783ba8c92dd33d508e469bc24eaacaab6. The current version is the one conduct all experiment. Another option is to download the docker image, you can find the instructions on github repo.

A.3.2 Hardware dependencies

The minimal configuration is at least 4 core CPU, 8G memory and at least 200G disk space. It's recommend to have more CPU cores, they will speedup the compiling, fuzzing and symbolic execution significantly. To run multiple cases at the same time, use `-pm` argument. We recommend you run `n` cases at the same time which n equals to the number of cores divide 4 (e.g, if you have 16 cores, we recommend you use `-pm 4` to run 4 cases that the same time)

A.3.3 Software dependencies

Software dependencies will be installed by running `python3 syzscope -install-requirements`. Or you can use the ready2go docker image within all dependencies installed

A.3.4 Data sets

https://docs.google.com/spreadsheets/d/16tt4Mo40iyWeuxOXBpRtmV_Zjdddta9nIy-poEuq66E/edit?usp=sharing

A.4 Installation

The detailed installation instructions are presented in the github repo https://github.com/seclab-ucr/SyzScope.

A.5 Experiment workflow

First, gather all cases we want SyzScope to run, copy the hash value from dataset page we provided, and paste them into a file, one hash per line. Second, run SyzScope with `nohup python3 syzscope -i dataset -RP -KF -SA -SE -timeout-kernel-fuzzing 3 -timeout-static-analysis 3600 -timeout-symbolic-execution 14400 -guided -be-bully &`, this process may take a long time. If you have more than 4 cores, you can run multiple cases at the same time by provide `-pm` arguments. For example, `nohup python3 syzscope -i dataset -RP -KF -SA -SE -timeout-kernel-fuzzing 3 -timeout-static-analysis 3600 -timeout-symbolic-execution 14400 -guided -be-bully -pm 6 &` runs 6 cases at the same time, but it requires at least 4*6 cores on your machines.

Third, cases that found high-risk impacts will be moved to `succeed` folder, cases that failed to find high-risk impacts will be moved to `completed` folders. Rerun those failed cases by using `-force`. For example, `python3 syzscope -i f99edaeec58ad40380ed5813d89e205861be2896 -RP -KF -SA -SE -timeout-kernel-fuzzing 3 -timeout-static-analysis 3600 -timeout-symbolic-execution 14400 -guided -be-bully -force`. If any error occurs, check out the common issues on our github page https://github.com/seclab-ucr/SyzScope/blob/master/tutorial/common_issues.md.

Final, check out the results from symbolic execution and compare it with the ones shown on our webpage https://sites.google.com/view/syzscope/home. The results of symbolic execution is in `work/succeed/xxx/sym-xxx/symbolic_execution.log`. The high-risk impacts stores under `work/succeed/xxx/sym-xxx/primitives`.

For example, to verify case `ce5f07d6ec3b5050b8f0728a3b389aa510f2591b`, you will find a function pointer dereference impact at `work/succeed/ce5f07d/sym-ori/primitives/FPD-try_to_wake_up-0xffffffff8137be7d-17` which related to the one we present on our webpage https://sites.google.com/view/syzscope/kasan-use-after-free-read-in-io_async_task_func.

A.6 Evaluation and expected results

Due to race condition, some bugs may be hard to trigger or trigger different contexts. We just need to run multiple times to increase the possibility of bug reproducing.

The final component is symbolic execution. To verify the final results from symbolic execution, check out the file "symbolic_execution.log". (Read more details at https://github.com/seclab-ucr/SyzScope/blob/master/tutorial/sym_exec.md) The number of new impacts shown at the end of the file.

In terms of the results, you may have slightly different output due to different configuration of experiment machine. We ran all experiments on Ubuntu-18.04 with 1TB memory and Intel(R) Xeon(R) Gold 6248 20 Core CPU @2.50GHz * 2. If you use machine that less powerful than ours, you might have less high-risk impacts comparing to our results. One approach to verify our results is through the CVE we obtained. We create a page to document the detailed analysis about the cases that received CVE, and each of them has at least one high-risk impact. These high-risk impacts are the reasons for CVE obtainment, so if you can verify those high-risk impacts on your end, it means the results are reproduceable.

See the link to each detailed analysis on our dataset page.

A.7 Notes

`457491c4672d7b52c1007db213d93e47c711fae6` has multiple UAF contexts due to race condition. Our web page shows only one of them(`ucma_close`), but another UAF context(`ucma_destroy_id`) may also lead to control flow hijacking.

`f99edaeec58ad40380ed5813d89e205861be2896` may be hard to trigger. If it failed to run symbolic execution, try `python3 syzscope -i f99edaeec58ad40380ed5813d89e205861be2896 -RP -SE -timeout-symbolic-execution 14400 -guided -force`.

`4bf11aa05c4ca51ce0df86e500fce486552dc8d2` has an arbitrary value write on a local variable in `hci_extended_inquiry_result_evt` shown on our webpage. However we abandoned any local variable write due to short life span of local variable and they are merely exploitable. So Running SyzScope on this case now will no longer find any high-risk impact.



A Artifact Appendix

A.1 Abstract

Our artifact is a pure software for fuzzing protocol implementations. It has no hardware requirement and very few software dependencies (Ubuntu Linux system, Python, and Docker). All of our approaches described in our paper are implemented in this artifact. The major claim is that our artifact is able to cover more of the state space. Without any manual intervention, the results can be found in artifact's execution log. We also provide scripts that may be used to reproduce the results.

A.2 Artifact check-list (meta-information)

- **Algorithm:** A new state approximation method for fuzzing protocol implementation.
- **Program:** We used the benchmark FuzzBench (https://github.com/bajinsheng/SGFuzz_Fuzzbench)
- **Compilation:** Clang >= 6.0
- **Run-time environment:** Ubuntu 20.04
- **Metrics:** State transition coverage, branch coverage
- **How much disk space required (approximately)?:** 20GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 2 days
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/bajinsheng/SGFuzz
- **Code licenses (if publicly available)?:** Apache License 2.0
- **Archived (explicitly provide DOI or stable reference)?:** Source code: https://github.com/bajinsheng/SGFuzz/tree/8f45141 Experimental data: https://zenodo.org/record/5555955

A.3 Description

A.3.1 How to access

git clone https://github.com/bajinsheng/SGFuzz
cd SGFuzz
git checkout 8f45141

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

- Ubuntu (>=16.04)
- Docker (>=20.10.7)
- Python (>=3.8)

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

We provide a script to compile and configure our artifact, and only two steps are needed to setup:

1. git clone https://github.com/bajinsheng/SGFuzz
2. cd SGFuzz && ./build.sh

A.5 Evaluation and expected results

We explain our major claims, corresponding results in the paper, and the detailed steps to reproduce them.

A.5.1 Key Claims

We made these key claims in our paper:

1. **State Transition Coverage**. SGFUZZ is able to cover more of the state space. SGFuzz significantly outperform LibFuzzer on state transition coverage in 23 hours.
2. **Branch Coverage**. SGFuzz slightly outperform LibFuzer on branch coverage in 23 hours.
3. **State Identification Effectiveness**. Changed variables with name constants are accurate approximation of state variables.
4. **Prevalence of Stateful Bugs**. Stateful bugs are prevalent in protocol implementations.
5. **Prevalence of State Variables**. Named constants are widely used for state variables.

A.5.2 Key Results

We have these key results in our paper to support our claims.

1. **State Transition Coverage**. SGFuzz covers 33x more sequences of state transitions than LibFuzzer in 23 hours on average. (Research Question 1)
2. **Branch Coverage**. SGFuzz achieves 2.20% more branch coverage than LibFuzzer in 23 hours on average. (Research Question 2)

3. **State Identification Effectiveness**. An average 99.5% of nodes in the State Transition Tree (the data structure constructed in SGFuzz for tracing states) in 23 hours are referring to values of actual state variables. (Research Question 3)

4. **Prevalence of Stateful Bugs**. Every four in five bugs that are reported in OSS-Fuzz for protocol implementations among our subjects are stateful. (Appendix Section 3)

5. **Prevalence of State Variables**. Top-50 most widely used open-source protocol implementations define state variables with named constants. (Appendix Section 4)

A.5.3 Prerequistities to reproduce

We have integrated our code into the FuzzBench framework, so only the dependencies of FuzzBench are necessary to evaluate our code. Please refer to the following commands to install and configure the FuzzBench.

```
git clone https://github.com/bajinsheng/
 → SGFuzz_Fuzzbench
cd SGFuzz_Fuzzbench
git submodule update --init
sudo apt-get install build-essential
 → python3.8-dev python3.8-venv
make install-dependencies
source .venv/bin/activate
```

More information about the installation of Fuzzbench can be found at: https://google.github.io/fuzzbench/getting-started/prerequisites/.

Note that the FuzzBench framework depends on docker, so it is hard to run FuzzBench within docker.

A.5.4 Steps to reproduce

1. **State Transition Coverage**.

(1) `SGFuzz's results.` Executing this command in the root of SGFuzz_FuzzBench folder:

```
sudo make run-sfuzzer-h2o_h2o-fuzzer-http2
```

After prompting some building information (several minutes for the first time), the fuzzing status will be gradually shown in the terminal, like this:

```
#2 INITED cov: 641 ft: 642 corp: 1/12569b
    exec/s: 0 rss: 38Mb states: 13 leaves: 2
#3 NEW    cov: 649 ft: 659 corp: 2/24Kb
    lim: 12569 exec/s: 0 rss: 39Mb states: 13
    leaves: 2 L: 12569/12569 MS: 1 CopyPart-
```

The number of *leaves* represents the number of unique state transition sequences observed in the current fuzzing campaign.

(2) `LibFuzzer's results.` As a reference, the results of LibFuzzer have to be got manually, because of the lack of *leaves* information. We copy the generated corpus from LibFuzzer to SGFuzz, and observe the *leaves* information.

Starting an interactive docker shell for LibFuzzer:

```
sudo make debug-libfuzzer-h2o_h2o-fuzzer-http2
```

In the docker container, running the LibFuzzer:

```
$ROOT_DIR/docker/benchmark-runner/startup-runner.sh
```

After 23 hours, in the docker container, typing 'CTRL+C' to stop LibFuzzer. In the host, copying the generated corpus from docker to host:

```
sudo docker cp docker-id-libfuzzer:/out/corpus
 → .
```

The *docker-id-libfuzzer* needs to be replaced by the actual hash id of the docker container. Then starting a docker container for SGFuzz:

```
sudo make debug-sfuzzer-h2o_h2o-fuzzer-http2
```

In the host, copying the corpus to the new docker container:

```
sudo docker cp corpus docker-id-sgfuzz:/out
```

The *docker-id-sgfuzz* should be replaced by the SGFuzz's docker hash id as well. In the SGFuzz's docker container, running SGFuzz to observe the results:

```
./h2o-fuzzer-http2 corpus/
```

In the output, the line with the **INITED** represents the total number of state transition sequences observed in LibFuzzer's campaign:

```
#1137 INITED cov: 1456 ft: 5274 corp: 375/2703Kb
    exec/s: 12 rss: 340Mb states: 235 leaves: 47
```

(3) `Evaluation.` Comparing the number of *leaves* indicated in each fuzzing campaign. Note that our experiments were conducted in 23 hours, so we may notice a substantial gap in state transition coverage between SGFuzz and LibFuzzer after **several hours**, not a few minutes.

(4) `More subjects.` Changing *h2o_h2o-fuzzer-http2* to *curl_curl_fuzzer*, *mbedtls_fuzz_dtlsserver*, *gstreamer_gst-discoverer* in the commands and redo steps 1-3 to evaluate other subjects.

(5) `Variance.` Our results are based on average number across 20 runs. Beware of variance! Difference between the two highest- and lowest-coverage runs may be up to 50% because of the randomness in fuzzing.

2. **Branch Coverage**.

The same steps as the state transition coverage experiment. The only difference is that the branch coverage information can directly got from the output of LibFuzzer, so we directly run

```
sudo make run-libfuzzer-h2o_h2o-fuzzer-http2
```

instead of the step (2) in State Transition Coverage. The branch coverage information is indicated as number of *cov* in the output.

3. **State Identification Effectiveness**.

Please check the folder *RQ3_State_Iden_Effic* at https://zenodo.org/record/5555955, which includes all state variables and the variables that are included in the STT.

4. **Prevalence of Stateful Bugs**.

Please check the folder *A3_Bug_Preva* at https://zenodo.org/record/5555955, which includes all state variables and the variables that are included in the STT.

5. **Prevalence of State Variables**.

Please check the folder *A4_Top50* at https://zenodo.org/record/5555955, which includes all state variables and the variables that are included in the STT.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

Our artifacts provide a program to identify files of different formats and pair them together if these formats makes it possible, as polyglot or optionally as near polyglots. Other programs we provide can then turn these files into ambiguous ciphertexts, and then decrypt these ciphertexts to verify the validity of the payloads.

Many examples of input files and ambiguous ciphertexts are also provided.

A.2 Artifact check-list (meta-information)

- **Program: Mitra, Key Commitment**
- **Compilation: Python 3 (Mitra), SageMath (Key Commitment)**
- **Output: [near] polyglots and ambiguous ciphertexts**
- **How much disk space required (approximately)?: 6 Mb**
- **How much time is needed to prepare workflow (approximately)?: 5 min**
- **How much time is needed to complete experiments (approximately)?: 5 mins**
- **Publicly available?: Y**
- **Code licenses (if publicly available)?: MIT**

A.3 Description

A.3.1 How to access

Complete source and examples are available at :

- the Mitra repository (`https://github.com/corkami/mitra/` at tag `Usenix22`) – for file manipulation and ciphertext generation with GCM.
- the Key Commitment repository (`https://github.com/kste/keycommitment/` at tag `Usenix22`) – for ciphertext generation with GCM, GCM-SIV and OCB3.

A.3.2 Software dependencies

Python 3 and SageMath with the Cryptography, PyCryptodome, PyMuPDF and BitVector packages.

A.4 Installation

Clone or download the repositories and install the extra packages if needed. Note that, depending on your SageMath installation, you might have to install the Python packages *within* SageMath, e.g.: `sage -python3 -m pip install pycryptodome`.

A.5 Evaluation and expected results

A.5.1 Ambiguous PDF/PE file

The PDF/PE combination was used because it can be included in the release of the IACR archive (`https://ia.cr/2020/1456`) – the article file itself is a proof of concept.

1. Generate a near polyglot

   From any PDF file, in `mitra/utils/extra` run `pdfpe.py <file.pdf> SumatraPDF18fixed.exe` (it works with other Windows executables – this one is provided as an example).

   You get a near polyglot file named like `Z(2-33-211420).exe.pdf`.

   Keep the file name intact, as it uses a special syntax to store extra information: in this case 2, 33 and 221420 are the offsets where the polyglot content switches to the other file type.

2. Generate an ambiguous ciphertext

   From the `Key Commitments` repository, run the following command :

   ```
   sage mitra_gcm.sage
     -k 4e6f773f00000000000000000000000000
        4c3474337221212100000000000000000
     -n 000000000000000000e7c6
     -a 4d79566f69636549734d795061737321
     "Z(2-33-211420).exe.pdf" -p > poly.gcm
   ```

3. Decrypt and validate the different payloads from the ambiguous ciphertext

   From the directory `mitra/utils/gcm`, run the `decrypt.py poly.gcm` command line.

   You obtain from the same ciphertext the original PDF document and a windows executable – that is a PDF viewer in this case.

   The files have changed of course, but they behave like the original input files.

A.5.2 Ambiguous HTML file

The HTML/HTML combination was used because it's tiny and demonstrates the vulnerability which can occur in a service like *Subscribe with Google* if an AEAD without key-commitment is used.

1. Use the following example files

   - `normal.htm` :

     ```
     <html>Hello World!</html>
     ```

- `evil.htm`:
  `<html><a href="http://www.evil.com">Click here!</a></html>`

2. Generate an ambiguous HTML file

   From `mitra/utils/extra`, run `htmhtm.py normal.htm evil.htm`, and you'll get a file called `(4-26)7.d3f286cd.htm.htm`.

3. Generate an ambiguous ciphertext

   in `mitra/utils/gcm`, run the following command:

   ```
   meringue.py
     -i 7  "(4-26)7.d3f286cd.htm.htm"
     attack.gcm
   ```

4. Validate the ambiguous ciphertext by extracting the different plaintexts

   From the `mitra/utils/gcm` directory, run `decrypt.py attack.gcm`.

A.5.3 Ambiguous JPEG/? file

This combination is mentioned because the brute-forcing was reduced to 4 bytes (as opposed to 6 bytes in prior works) and requires a special workflow with post-processing of the near polyglot.

1. Generate a (non-working) near polyglot

   Use with any file that is supported with JPEG near polyglots, for example ICC files.

   ```
   mitra.py <file.jpg> <file2.bin> --verbose
   --overlap
   ```

   If ran on a JPEG, it will output the following warning :

   ```
   > Jpeg overlap file: reducing two bytes
   > (don't forget to post-process
       after bruteforcing)
   ```

   and generate a near polyglot file.

2. Find a valid nonce for the near polyglot

   From `mitra/utils/gcm/`, run the command `nonce.py <near_polyglot>`.

   This bruteforcing operation will use the 2-bytes shortcut but requires extra post-processing. This script will output a nonce value.

3. Post-process the near polyglot

   Run `jpg4fix.py <near_polyglot> <nonce>`. It will generate a fixed near polyglot starting with `4-`.

4. Generate the ambiguous ciphertext

   Run the following command:

   ```
   meringue.py
    -k 01010101010101010101010101010101
       02020202020202020202020202020202
    -i <index_offset>
    -n <nonce>
    <fixed_near_poly>
    ambiguous.gcm
   ```

   This execution will generate an ambiguous ciphertext.

   The index argument is the block index of the file where the TAG will be written. The indexed block should be blank : if the near polyglot file doesn't have such a block, you might want to add appended data to the polyglot itself or to the parasite inside.

5. Generate and validate the different payloads

   In the `mitra/utils/gcm/` folder, run the following `decrypt.py ambiguous.gcm` command.

   Once again, you'll get different files that just work like the input files.

A.6 Experiment customization

These file operations will work with different cryptographic parameters (keys, nonces, index), and other block ciphers with reasonable code modifications.

Many other file formats are supported by Mitra and can be combined as polyglots or near polyglots. Make sure you use **standard input files**: weird files created by Mitra might not be supported by Mitra itself as they may not have a standard structure anymore – you may want to use the `input/*` files provided in Mitra as a start.

Use the `--verbose` flag to get more feedback (e.g. why a polyglot was not generated). Use the `--reverse` flag if you're not sure which files should come first and last in the command line.

Other block cipher modes such as **OCB and GCM-SIV are supported** and included in the Key Commitment repository.

Use the `mitra_ocb.sage` or `mitra_siv.sage` scripts to generate ambiguous ciphertexts, and the `decrypt_ocb.sage` or `decrypt_siv.sage` scripts accordingly to decrypt payloads.

Unlike other modes that have byte granularity, GCM-SIV and OCB3 require **block alignment**. You may want to add two blocks of pre-padding and post-padding to the parasite when generating the [near] polyglot files.

For GCM-SIV, the **complexity** of generating an ambiguous ciphertext requires solving a system of linear equations of size relative to the files size, while it's constant for the other modes. The other expensive operation is bruteforcing the nonce of ambiguous ciphertext from near polyglots, which depends on the size of the overlap.





A Artifact Appendix

A.1 Abstract

This artifact contains the C++ source code of our novel DP-PIR protocol that we introduce in the paper. Our protocol allows clients to privately query the contents of a remote database, without revealing information about their query to the service beyond a well-defined differentially private leakage. Unlike previous PIR protocols, DP-PIR amortizes queries from independent clients leading to constant amortized server and client side computation and communication complexity when the query volume is sufficiently large with respect to the size of the database.

Our artifact includes the code for the different entities of our protocol. This includes the client(s) code, as well as the two or more parties that constitute the service. In addition, the artifact includes scripts to run the various experiments and produce the plots and tables we show in the paper, including the comparisons with existing three existing protocols: Checklist, DPF, and SealPIR. Our implementation uses Google's Bazel build system, and includes Bazel ports for building the three aforementioned baselines. We developed our implementation using Ubuntu 20.04 and g++-11. We will provide a Docker container with all required dependencies by the artifact submission deadline.

The primary purpose if this artifact is to (1) support the claims of our paper about the efficacy of amortization with DP-PIR compared to current state of the art protocols, and (2) demonstrate how the performance of DP-PIR is governed by the different application parameters. To that end, we designed and ran several experiments that run our implementation or the baselines with different parameters and report the total service time taken to process the generated query loads. We ran our experiments on AWS instances to mimic a realistic setup where the different parties making up the protocol are deployed over separate machines and communicate over realistic networks. Most experiments can be run locally without AWS, except a couple of the larger data points that will most likely run out of memory when all the parties are run on the same local machine. We will provide detailed instructions on how to run experiments locally or over AWS, and how to interpret and plot the results by the deadline.

A.2 Artifact check-list (meta-information)

- **Algorithm:** This artifact provides an implementation of DP-PIR, a new private information retrieval protocol.
- **Compilation:** We tested our artifact using G++-11. Our implementation uses Google's Bazel build system.
- **Binary:** No binaries are included. The protocol binary should be build from the source code using Bazel in optimized mode.
- **Run-time environment:** We developed our artifact on Ubuntu 20.04. We will provide a docker container with all the relevant dependencies.
- **Run-time state:** Our implementation, and especially the online portion of our protocol, is extremely sensitive to network bandwidth. In our experiments, we deployed our AWS instances in a cluster placement group to minimize network costs.
- **Execution:** Each data point on any of the plots in the paper is a separate running job spawned by our scripts. For the smaller data points, the execution takes a few seconds, but for the larger ones (e.g. 100M queries), it may take close to an hour.
- **Security, privacy, and ethical concerns:** There are no such concerns.
- **Metrics:** We report the total service side execution time. Concretely, this is the wall time between the first party/server in the protocol receiving the last query in the batch, right before processing of the batch starts, and the wall time at that same server right after the batch has been processed, and before the responses are sent to client(s). We report similar measurements for the baselines as well.
- **Output:** The experiments produce log files for the different parties, each file containing various debugging information as well as time measurements. Our artifact includes scripts that automatically process these files to extract the relevant information, and produce plots similar to the ones shown in the paper.
- **Experiments:** Our experiments are run via an "orchestrator" command line program provided in the artifact. This orchestrator is a simple nodejs web server that workers (local or AWS) ping for jobs. Aside from running these workers once, evaluators need only interact with the orchestrator via its control interface, e.g. they can use command 'load figure1' to direct the orchestrator to load and run the experiments needed to produce figure 1 in the paper.. The orchestrator is responsible for translating input commands into jobs, assigning them to workers, and tracking the progress of these workers including acquiring their output files.
- **How much time is needed to complete experiments:** In our setup, the experiments require about 14 hours of mostly *passive* running time to produce the results shown in the paper.
- **Publicly available:** at https://github.com/multiparty/DP-PIR/tree/usenix2022.
- **Code licenses:** MIT

A.3 Description

A.3.1 How to access

Clone this repository https://github.com/multiparty/DP-PIR/tree/usenix2022.

A.3.2 Hardware dependencies

We ran our experiments using one r4.xlarge AWS instance per server/party. These instances have 2 vCPUs and 30.5GB RAM. If run locally, more memory will be required to run the larger experiments, since all the parties (and thus all their memory) will be run

on the same machine. In such cases, we recommend that proportionally smaller experiments are run to fit the hardware constraints. The artifact documentation includes more details on this.

A.4 Installation

We provide a Docker container that includes all require dependencies. Instructions on building and running this container are provided in the artifact. We also provide instructions on how to deploy and run the experiments locally or via AWS.

A.5 Experiment workflow

To simplify running experiments, we provide an orchestrator program included in the artifact. The orchestrator takes care of configuring the protocol per the experiment parameters. The orchestrator is ideal for experiments with many parties or parallel machines, as it automatically assigns the experiment tasks to the workers and monitors their progress.

At a high level, our workflow with the orchestrator follows these steps:

1. The orchestrator application is run via the command line.
2. Several workers are spawned, either as AWS instances or locally via the provided scripts. As many workers are needed as the sum of parties and clients. For most experiments in the paper, this translates to 3 workers needed for 2 parties and 1 client.
3. The workers execute background daemon scripts that periodically ping the orchestrator to request jobs or report progress.
4. Evaluators issue commands to the orchestrator to run instances of our protocol with specific parameters. All the parameters for all of the results in the paper are packaged inside the artifact and can be loaded by name (e.g. 'load figure1'). However, evaluators can also run experiments with different parameters, which they need to specify to the orchestrator via an interactive dialog.
5. The workers receive the jobs corresponding to the different parameters issued by the evaluator. Workers run these jobs, which include running various steps of the protocols, such as creating queries, shuffling queries, and exchanging various messages over the network. Whenever a worker finishes a job, it reports that to the orchestrator along with the output file, which include the measured computation time.
6. The orchestrator notifies evaluators whenever workers and experiments are completed. The evaluators can then run the plotting script provided in the artifact to plot the results similar to the plots in our paper.

It is possible to run experiments directly using the protocol implementation without relying on the orchestrator. Consult the artifact documentation for more details.

A.6 Evaluation and expected results

The main goal of this artifact is to produce plots showing the performance of DP-PIR as a function of the different application and setup parameters. Specifically, we are interested in demonstrating (1) how the performance of DP-PIR compares to that of existing protocols, and (2) how the performance of DP-PIR scales with different parameters.

With our protocol, we have the following parameters:

1. The database size: the number of rows in the database being privately queried. In our figures, this size ranges between 2.5 million rows for the larger experiments and 100K or 10K rows for the smaller ones.
2. The number of queries: how many queries to process via the protocol. This can range between several thousands and hundreds of millions. Note: on setups with limited RAM, the artifact will not be able to handle the larger query numbers as it will run out of memory.
3. The number of parties: how many parties constitute the service. Our protocol requires at least two parties and tolerates up to $n-1$ malicious and colluding parties. This is almost always set to 2 in our experiments, except figure 5 where it ranges between 2 and 5.
4. Parallelism: how many instances/workers/servers does a party possess. The more servers here the faster the protocol will run as the queries get split among these servers. We almost always set this to 1 except in table 2.
5. ε and δ: the differential privacy parameters governing how much leakage is tolerated. Smaller parameters imply more privacy at the cost of performance.
6. The mode: whether we are measuring the offline or online portions of our protocol.

When creating a job, the orchestrator will interactively request these parameters from the evaluators. Alternatively, the orchestrator can be instructed to load one or more bundled experiment which specifies all these parameters in accordance to the paper.

The main expected result here is a confirmation of the efficacy of our protocol and its amortization. Specifically, that our protocol becomes significantly faster than existing systems as the number of queries approaches or exceeds the database size. This is demonstrated by producing a plot similar to figure 1 in the paper: all the parameters are fixed (in the paper: DB size = 2.5M, parties = 2, parallelism = 1, $\varepsilon = 0.1$, $\delta = 10^{-6}$), while varying the number of queries (e.g. from 10^4 to 10^8). For each number of queries, we run both our *online* protocol and an existing PIR protocol (e.g. Checklist), and plot the reported runtimes as a function of the number of queries. We shows our results from the paper for demonstration below.

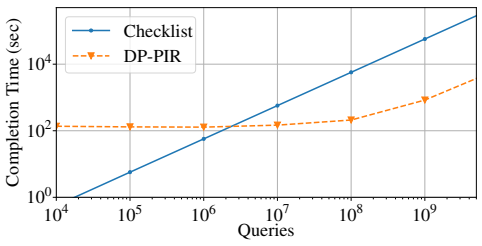


Figure 1: Checklist and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 2.5M database

The exact numbers shown in the plot are setup dependent, and may significantly vary between setups. Our protocol is extremely susceptible to any changes in network bandwidth and latency. However, we expect to see three trends: (1) The total runtime of checklist is proportional to $O(|queries| \times \sqrt{|DB\ size|}$. (2) DP-PIR runtime initially is constant and does not seem to grow much with the number of queries. As the number of queries becomes similar in magnitude to the database size, our performance starts to grow with the number of queries. (3) Our protocol is (much) slower than Checklist for few queries, and much faster than Checklist for huge number of queries. Checklist's graph crosses over DP-PIR's somewhere in the middle, for a number of queries roughly in $O(|DB\ size|)$. A reasonable number of queries would be between 0.5 to 2.5 times the database size, depending on the setup.

If these three trends are observed, then the result match our expectations and confirms our claims about the performance of DP-PIR and its amortization. If either of them is absent, specifically, if DP-PIR remains slower than or comparative to Checklist even as the number of queries becomes large, that would disprove our performance and efficiency claims.

Another expected result is to demonstrate that DP-PIR scales with the different parameters as expected. Specifically, that it scales linearly in the number of queries and database size, for both online and offline stages, scales super-linearly in the number of parties in the offline stage, and exhibits close to linear speedups when additional parallel resources are used. These can be validated by fixing all the parameters except the parameter under investigation, and plotting the performance of DP-PIR as a function of that singular parameter. The produced plots should exhibit similar trends to the plots in section 7 of the paper.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

This appendix describes the software artifact that implements and evaluates all algorithms proposed in this paper. Specifically, it provides Java implementations for the *Greedy*, *SortGreedy* and *Hungarian* algorithms for anonymization by both ℓ-diversity and β-likeness; it also contains implementations of the algorithms we compare against, namely NH (in C++), BuReL (in Java) and PrivBayes (in C++), as obtained by their authors and properly enhanced to record the same metrics.

No specialized hardware is required to reproduce the results of the paper; however, the anonymization of the largest dataset requires at least 64GB of RAM. We provide instructions on how to install the artifact, execute the experiments, and validate the results in the form of a README document that describes the process step by step. This is intended to help the reader reproduce the results presented in the paper. The artifact is available as a GitHub repository.

A.2 Artifact check-list (meta-information)

- **Algorithm:** We present three novel algorithms for disclosure control through syntactic anonymization based on the notion of heterogeneous generalization. These algorithms are (i) *Greedy*, denoted as GR, which employs an $O(n^2)$ heuristic for assignment extraction, (ii) *SortGreedy*, denoted as SG, which employs an $O(n^2 \log n)$ heuristic for tuple matching and (iii) *Hungarian*, denoted as HG, which utilizes the $O(n^3)$ Hungarian algorithm to build assignments. These algorithms are customized to both ℓ-diversity and β-likeness.
- **Data set:** We use real and synthetic datasets of up to 500k tuples and 8 dimensions. All data are included in the repository.
- **Run-time environment:** Our artifact is not OS-specific. However, all experiments have been performed on an Ubuntu 16.04 LTS server with jre 1.8.0_11. Perl scripts are provided for batch experiment submission (perl v5.20.2). No root access is required.
- **Hardware:** No special hardware is required. However the anonymization of the largest dataset (500k tuples) requires at least 64GB of RAM.
- **Execution:** The anonymization of the largest dataset (500k) may take 3-4 days to complete. Most of the experiments are performed with the default dataset of 10k tuples and each of them lasts up to a couple of minutes.
- **Security, privacy, and ethical concerns:** Since only synthetic as well as open and publicly available datasets have been used, there are no security, privacy, or ethical implications in running the experiments.
- **Metrics:** The evaluation metrics include execution time, information loss incurred by the anonymization, accuracy of queries and attacks on the anonymized dataset.
- **Output:** The output is printed in text (.txt) files, following a specific format.
- **Experiments:** All experiments can be replicated and results reproduced simply by cloning the repo, compiling the code (C++ for NH and PrivBayes, and Java for all the rest) and running the perl scripts provided - one for each of the experiments - following the instructions in the README file.
- **How much disk space required (approximately)?:** In the order of MB.
- **How much time is needed to prepare workflow (approximately)?:** A few minutes.
- **How much time is needed to complete experiments (approximately)?:** 4-5 days.
- **Publicly available (explicitly provide evolving version reference)?:** Our artifact, excluding the source code of the NH, BuReL and PrivBayes algorithms that we compare against, are publicly available under an open source license.
- **Code licenses (if publicly available)?:** Apache Licence 2.0.
- **Archived (explicitly provide DOI or stable reference)?:** (https://github.com/discont/disclosurecontrol/releases/tag/artifact-evaluation).

A.3 Description

A.3.1 How to access

The artifact is publicly available and hosted by GitHub here:

https://github.com/discont/disclosurecontrol

To download the latest version, clone the repository using the command

```
git clone https://github.com/discont/disclosurecontrol.git
```

A.3.2 Hardware dependencies

No specific hardware features are required to evaluate the artifact. To be able to execute our algorithms using the largest input dataset at least 64GB of RAM are required. For the default dataset of 10k tuples, used in the majority of experiments, 8GB of RAM will suffice. As for disk space, our artifact has minimal requirements of a few MB.

A.3.3 Software dependencies

All experiments ran on an Ubuntu 16.04 LTS server. Java code has been compiled with jdk1.8.9_291. Perl scripts were used for batch experiment submission (perl v5.20.2).

A.3.4 Data sets

We use real data drawn form the CENSUS and the COIL 2000 datasets, which are publicly available. Additionally, we generate synthetic datasets of up to 500K tuples and 8 attributes based on the CENSUS data, varying the bias of the sensitive value distribution. All datasets are included in the repository.

A.4 Installation

First, clone the relevant GitHub repository. Compile the source code using plain javac or your favorite Java IDE. Place the .class files in a folder within the local directory where the GitHub repo has been cloned, named bin. To reproduce the experiments, use the provided perl scripts, along with the input datasets. The README file offers a step-by-step guide.

A.5 Experiment workflow

All experiments can be executed by invoking the relevant perl scripts as described in the accompanying README file.

A.6 Evaluation and expected results

Our experiments start by evaluating the application of our algorithms on the achievement of ℓ-diversity. Our findings show that our methods outperform the state-of-the-art NH in utility under various values of the privacy parameter ℓ, the number of *QI* dimensions d, the dataset size n and the skewness of the data distribution (Figures 2, 3, 4a, 4c, 5a and 5c). Our schemes can be applied on partitions of the input dataset in a data-parallel environment, slightly sacrificing utility for the sake of scalability (Figures 4a, 4b, 5a and 5b).

Then, we adapt our algorithms to achieve β-likeness. The experiments demonstrate that our methods offer better utility than the state-of-the-art β-likeness algorithm, BuReL, regardless of the β value and the distribution of tuple values (Figure 6). The utility gain of our schemes compared to BuReL grows with data size (Figure 7a) but shrinks with the skewness of dataset values (Figure 7c). In all cases, at least one of our methods outperforms BuReL in terms of utility. Moreover, our algorithms provide anonymized datasets that can serve range and prefix queries of various selectivities with significantly better accuracy — in terms of median query relative error — compared to BuReL and PrivBayes (Figures 8 and 9). Last, our schemes provide stronger resistance than state-of-the-art differential privacy schemes (PrivBayes) to learning-based attacks under our adversary model on real-world data (Figure 11).

Each of the aforementioned results can be reproduced by simply running automated perl scripts accompanying the code. The scripts execute the provided code with the necessary parameters and record the metrics of interest. The README file walks the user through this process.

Due to the intentional introduction of randomness in the tuple assignment extraction stage, information loss results may slightly differ in each run. The same applies to the accuracy of range and prefix queries, which are randomly generated at run time. Query accuracy may exhibit larger deviations, thus we suggest to execute relevant experiments multiple times and adopt the median value.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

In our paper we explore how users consider security and privacy in light of third-party API access to their Google accounts given the disclosure and control mechanisms currently available. First, we surveyed n = 432 participants to recall the last times they used Google SSO or authorized a third-party app access to their Google account data. We then invited n = 214 participants from the first survey to return for a follow-up survey. As part of this second survey, participants installed a browser extension that parsed entries in their Google account's "Apps with access to your account" dashboard. In our archive we make available functional artifacts that can be used to reproduce our qualitative and quantitative study results. The artifact includes the custom survey and browser extension software we developed for this study along with detailed instructions on how to deploy this software. A single PC, Mac, or Linux machine should be sufficient hardware. Software requirements include Docker, RStudio, and a Chrome or Firefox web browser. The artifact can be evaluated by running the survey software in a docker container, loading the browser extension in a web browser, running the R-programming files, and evaluating the qualitative coding results.

A.2 Artifact check-list (meta-information)

- **Data set:**
  Provided in the survey-data folder of the GitHub archive.
- **Run-time environment:**
  Docker, R-Studio
- **Hardware:**
  a single PC, Mac, or Linux machine.
- **How much disk space required (approximately)?:**
  100 MB
- **How much time is needed to prepare workflow (approximately)?:**
  2 hours
- **How much time is needed to complete experiments (approximately)?:**
  4 hours
- **Publicly available (explicitly provide evolving version reference)?:**
  https://github.com/gwusec/api-privacy-archive-usenix22/
- **Archived (explicitly provide DOI or stable reference)?:**
  https://github.com/gwusec/api-privacy-archive-usenix22/tree/116104e7112b311ccf0567b58aebaf3f13ee3a50

A.3 Description

This archive contains the software developed for this API privacy research, and the data obtained from an online survey conducted during our study. The archive includes qualitative open coding analysis of open-ended survey results, as well as the r-scripts used to processes the quantitative results. We have included all of the software that we created to deploy the online survey, and the code for a custom browser extension that we built that works along with the survey. We have provided instructions for running this software in a docker container.

A.3.1 How to access

The artifact can be accessed at the following URL:
https://github.com/gwusec/api-privacy-archive-usenix22/tree/116104e7112b311ccf0567b58aebaf3f13ee3a50

Please read the provided README.md file for full details: https://github.com/gwusec/api-privacy-archive-usenix22/blob/116104e7112b311ccf0567b58aebaf3f13ee3a50/README.md

A.3.2 Hardware dependencies

A single PC, Mac, or Linux machine should be sufficient hardware.

A.3.3 Software dependencies

Software requirements: Docker, RStudio, Chrome or Firefox web browser.

A.3.4 Data sets

https://github.com/gwusec/api-privacy-archive-usenix22/tree/116104e7112b311ccf0567b58aebaf3f13ee3a50/survey-data

A.4 Installation

https://github.com/gwusec/api-privacy-archive-usenix22/blob/116104e7112b311ccf0567b58aebaf3f13ee3a50/survey/README.md

A.5 Experiment workflow

We created and deployed two online surveys. We collected the data from the surveys and used qualitative open coding to analyse the qualitative results, and R-programming to analyse the quantitative results. Both the qualitative spreadsheets and R scripts are provided, along with the raw data from the surveys.

A.6 Evaluation and expected results

The key results of the paper are the online survey result data and the detailed analysis of this data. The software that we developed allowed us to collect this data and this software can be validated. The figures and regression analysis as described in the paper is our next key results, and those can be validated using the raw data along with the provided R-scripts.

A.7 Experiment customization

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

A Artifact Appendix

A.1 Abstract

The artifact is composed of 3 directories. The Blockchain directory contains the implementation of Sawtooth blockchain, including smart contracts and 3 entities implemented also in containers. They're namely the administrator (manufacturer in the paper) and 2 clients with roles of Verifer and Prover. The code is deployed using docker and docker-compose. Deploying this part requires a Linux server with 24GB RAM and allows to verify the workflow of the proposed attestation scheme in our paper. The IoT-Clients directory containes the implementation of IoT client in 2 boards (LPCXpresso55S69 + Mikroe WiFi 10 Click and Atmel MEGA-1284P Xplained). The implementation is written in C. The installation and verification of the code helps confirm the low overhead of the attestation process. The Simulation directory contains the code used in the scalability evaluation. It is implemented in Python and helps evaluate the scalability of our proposed scheme against LegIoT.

A.2 Artifact check-list (meta-information)

Obligatory. Fill in whatever is applicable with some keywords and remove unrelated items.

- **Compilation:** AVR-Toolchain, arm-none-eabi, SDK_2.x_LPCXpresso55S69 [API version=2.0.0, Format version=3.8]
- **Run-time environment:** Ubuntu 18 with sudo rights
- **Hardware:** server with 24GB RAM and 16 CPUs, LPCXpresso55S69 + Mikroe WiFi 10 Click and Atmel MEGA-1284P Xplained. They are all publicly available.
- **Output:** console
- **Experiments:** manual steps for each experiment are provided in README file in each directory.
- **How much disk space required (approximately)?:** 20GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 3 hours
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/sss-wue/scraps
  Release tag 1.0.2-beta
- **Code licenses (if publicly available)?:** Licensed under the Apache License, Version 2.0
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/sss-wue/scraps/releases/tag/1.0.2-beta

A.3 Description

A.3.1 How to access

The source code is publicly available at https://github.com/sss-wue/scraps/releases/tag/1.0.2-beta

A.3.2 Hardware dependencies

The deployment of the blockchain network requires a server with at least 24GB RAM and 16 CPUs. Experiments with the IoT clients require installing the code on the respective boards: LPCXpresso55S69 + Mikroe WiFi 10 Click, Atmel MEGA-1284P Xplained and Atmel ICE.

The artifact is roughly 32MB of size. The deployment requires downloading and running several docker containers. 20GB of disk is sufficient for the deployment and experimenting.

A.3.3 Software dependencies

The deployment of the blockchain requires docker engine and docker compose to be installed. Installing the IoT clients on the boards requires MCUXpresso IDE v11.4.1 [Build 6260] [2021-09-15] and avrdude.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

In order to experiment with the artifact, it is only required to install the compilers and the software dependencies. Each of the boards needs to be connected to a computer using their debugging ports. LPCXpresso55S69 can be linked using a normal mini usb cable while Atmel MEGA-1284P Xplained is linked using Atmel ICE programmer.

A.5 Experiment workflow

The first step in the experiments is the deployment of the blockchain network. A YAML file is provided to automate the process using docker compose. Instructions on executing attestation process are handed in details in the respective README file.

Compiling and installing the IOT Client in LPCXpresso55S69 board is achieved using MCUXpresso software. A video is included in the respective directory showing the steps and the expected results.

Compiling and installing the IOT Client in LPCXpresso55S69 is automated using a Makefile. Used instructions and expected results are all documented and provided in the respective README file.

The simulation is executed using a Python script. Commands and examples of the results are explained in README file.

A.6 Evaluation and expected results

In our paper, we present a blockchain based attestation scheme for IoT devices. Experimenting with the blockchain network allows examining the workflow of the attestation scheme. Different entities are deployed in the roles described in our paper, such as Manufacturer, Prover and Verifier.

Moreover,installing and running the IoT clients on the respective boards prove that COTS IoT devices can benefit from our proposed scheme without any hardware modification.

Besides, running the simulations with different parameters show how our scheme outperforms the state-of-art scheme LegIoT.

A.7 Experiment customization

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

The most important experiment for our work is the searching for programs experiments described in Section 5.1. This experiment involves a binary compiled from C++ source code called `autoda`. It requires about 20 CPU cores and one GPU with at least 8 GiB of GPU memory to run efficiently. The ablation study experiments described in Section 5.4 is also a binary compiled from C++ source code, and have similar requirement with the `autoda` binary. The benchmark experiments requires GPUs. Its entry point is a python script.

The whole experiments in Section 5.1 and Section 5.4 should spend about one hundred GPU hours. The whole benchmark experiments in Section 5.2 and Section 5.3 should spend about several hundreds GPU hours. The searching for programs experiments can hardly be exactly reproduced, since randomly generating exactly the same program is extreme rare. However, the lowest ℓ_2 distortion ratio in each run can be reproduced. What's more, we provide the log files where we found the AutoDA 1st and 2nd in the `log/` directory.

A.2 Artifact check-list (meta-information)

- **Program:** See the `README.md` file.
- **Compilation:** See the `README.md` file.
- **Binary:** The `autoda` binary is for Experiment 5.1, and the `autoda_ablation` binary is for Experiment 5.4.
- **Model:** Most of them are pre-trained models provided by previous work, see `README.md`. The DenseNet, DPN, and DLA models are trained by ourselves using public available code at https://github.com/kuangliu/pytorch-cifar.
- **Data set:** The CIFAR-10 dataset and ImageNet dataset. The ImageNet dataset requires pre-processing, see `prepare_models/README.md`.
- **Run-time environment:** GNU/Linux. Need root access to install necessary dependencies.
- **Hardware:** Need GPU with at least 8 GiB of GPU memory. Need CPU with at least 20 cores.
- **Execution:** In our searching experiments described in Section 5.1, we run the same binary `autoda` for 50 times, and each run spends about two hours with one GTX 1080 Ti GPU and about 20 CPU threads. The full benchmark experiments in Section 5.2 and Section 5.3 should spend about several hundreds GPU hours.
- **Metrics and Output:** For the `autoda` binary, it would write the program we found into the log file passed via command line, as well as evaluation metrics (lines started with `rs=`, you could get evaluation metric for each programs by parsing the floating-point number in these lines). Since our searching for programs experiments described in Section 5.1 cannot be exactly reproduced, we provide the two runs' log files in the `log/` directory where the AutoDA 1st and 2nd algorithms are found. For the `autoda_ablation` binary, it would write the ratios into the log file passed via command line (lines started with `ratios_mean=`).
- **Experiments:** Besides the searching for programs experiments and the ablation study experiments, we also have benchmark experiments (Section 5.2 and Section 5.3). These benchmark experiments all use the `prepare_models/attacker.py` script. See `README.md` for more details on the usage of this script. This script would output adversarial example's distance to the original image at each step for each sample into a hdf5 file, thus further analysis could be done. This script would also print out the attack success rate and average/median ℓ_2 distortion during the attack process for the current batch.
- **How much disk space required (approximately)?:** 20 GiB.
- **How much time is needed to prepare workflow (approximately)?:** Setting up environment for compiling the `autoda` binary and the `autoda_ablation` is relative easy, several hours should be enough. Setting up environment for benchmark experiments from scratch are much more complicated, including training some models from scratch, downloading and pre-processing ImageNet dataset, and installing dependencies for every model.
- **How much time is needed to complete experiments (approximately)?:** About one hundreds GPU hours for the search for programs experiments and the ablation study experiments. About several hundreds GPU hours for benchmark experiments.
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/Fugoes/AutoDA/commit/257cf85e1c0c1d129a50a274764ed6bc893ccde5.
- **Code licenses (if publicly available)?:** MIT.

A.3 Description

A.3.1 How to access

https://github.com/Fugoes/AutoDA/commit/257cf85e1c0c1d129a50a274764ed6bc893ccde5.

A.3.2 Hardware dependencies

Need Nvidia GPU with at least 8 GiB of GPU memory. Need CPU with at least 20 cores.

A.3.3 Software dependencies

See `README.md`.

A.3.4 Data sets

CIFAR-10 and ImageNet.

A.3.5 Models

See `README.md`.

A.3.6 Security, privacy, and ethical concerns

This work does not have security, privacy, or ethical concerns.

A.4 Installation

See `README.md`.

A.5 Experiment workflow

For experiments in Section 5.1 and Section 5.4, first compile the source code:

```
cd␣~/
git␣clone␣git@github.com:Fugoes/AutoDA.git
cd␣AutoDA/
mkdir␣build/
cd␣build/
cmake␣-DCMAKE_BUILD_TYPE=Release␣~/AutoDA
make␣-j`nproc`
```

These commands will build the `autoda` binary and `autoda_ablation` binary.

To run the `autoda` binary,

```
CUDA_VISIBLE_DEVICES=0␣./autoda␣\
␣␣␣␣--dir␣~/path/to/data␣\
␣␣␣␣--threads␣16␣\
␣␣␣␣--gen-threads␣20␣\
␣␣␣␣--class-0␣0␣--class-1␣1␣\
␣␣␣␣--cpu-batch-size␣150␣\
␣␣␣␣--gpu-batch-size␣1500␣\
␣␣␣␣--max-queries␣500000000␣\
␣␣␣␣--output␣5ww_queries_00.log
```

The evaluation metrics would be written to `/path/to/data/5ww_queries_00.log`. To quickly check the lowest ℓ_2 distortion ratios,

```
cat␣5ww_queries_00.log␣|
␣␣␣␣grep␣rs=␣|
␣␣␣␣sort␣|␣head
```

Running the `autoda_ablation` binary is similar to running `autoda`. To quickly check for the top 200 lowest ratios,

```
cat␣ablation.log␣|
␣␣␣␣grep␣'^ratios_mean='␣|
␣␣␣␣awk␣-F␣'='␣"{␣print␣\$2␣}"␣|
␣␣␣␣sort␣-n␣|␣head␣-200
```

As for the benchmark experiments, please check the `README.md` file.

A.6 Evaluation and expected results

Though all attacks have randomness, when running the benchmark experiments, the results should be quite close to our reported results given large enough test set.

A.7 Experiment customization

A.8 Notes

The `autoda` and `autoda_ablation` does not exit cleanly, they would crash themselves when reaching max queries and core dumped.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





G Artifact Appendix

G.1 Abstract

The artifacts consist of various cache-timing experiments. The initial experiments help to understand the non-inclusive LLC structure and DDIO-based accesses to it. The later experiments allow reverse-engineering the underlying mechanisms, exploiting the findings and performance analysis.

As our work explores the use of FPGAs for cache side-channel attacks, the artifacts require an FPGA acceleration card. Though we used our local setup with Intel Programmable Accelerator Card (PAC), we also tested the artifacts with remote access to Intel Labs (IL) Academic Compute Environment (ACE) (https://wiki.intel-research.net/FPGA.html).

For people who have neither, we provide detailed documentation for each experiment, which provides the expected execution output and the inferences we draw from them. **Notably, Figures 3, 4, 7, 8 and 9 were produced based on experiments like these.**

G.2 Artifact check-list (meta-information)

- **Algorithm:** SW+HW Combined Cache Attacks, Eviction Set Construction, Reverse-Engineering DDIO
- **Compilation:** `make`, `gcc` (tested versions are 7.5.0 and 4.8.5)
- **Binary:** For software: binaries are not provided. Not needed as compilation is straightforward. For hardware: A bitstream file synthesised for Intel Arria 10 PAC is provided. This should save quite some time as synthesis may take several hours. Instructions on how to synthesise for other PACs are available.
- **Run-time environment:** `Ubuntu 18.04.5 LTS` and `CentOS Linux 7.7.1908` The basic experiments do not require any root access. However, experiments on non-default system configurations and reverse engineering require `sudo`. We included these experiments as they were requested by our reviewers.
- **Hardware:** Intel PAC (Programmable Accelerator Card)
- **Execution:** Compilation and execution of binaries on command line.
- **Security, privacy, and ethical concerns:** Demonstrates cache attacks allowing to steal secrets of victims sharing the same computer as attacker.
- **Metrics:** Cache access timings, execution time, accuracy.
- **Output:** Console output with exact numerical results.
- **Experiments:** The repository is a collection of various experiments, such as eviction set construction, cache access timings, eviction candidate determination, effect of shared access over the eviction candidate, evidence for DDIO$^+$ region, etc. A separate documentation file is provided for each experiment.
- **How much disk space required:** The repository is 13 MB. When de-compressed, the bitstream file reaches has a size of 133 MB. All in all, it is less than 150 MB.
- **How much time is needed to prepare workflow:** Almost no additional preparation is needed if you already have access to an Intel PAC0based FPGA-accelerated computation server. Otherwise, an access request to *Intel Labs Academic Compute Environment* (https://wiki.intel-research.net/FPGA.html) and getting a response might take at least a day.
- **How much time is needed to complete experiments:** It can take several hours.
  Suppose evaluators do not have access to an Intel PAC based FPGA accelerated computation server. In that case, they can go through the documentation files and observe the expected execution results and the inferences drawn from them. This option would take an hour.
- **Publicly available:** https://github.com/KULeuven-COSIC/Double-Trouble
- **Code licenses:** MIT License
- **Archived:** https://github.com/KULeuven-COSIC/Double-Trouble/tree/ArtifactsAvailable

G.3 Description

The repository contains a set of experiments, listed in the following table along with the relevant section of the paper.

Experiment	Figure/Section
Basic Functionality	
Shared Access	Figure 4d
CPU Read	Figure 4f
Secondary Write	Figure 4e
CPU Write	Figure 4c
Cache Timing Histogram	Appendix A
Eviction Candidate	Section 3.2.2
DDIO Replacement Policy	Appendix C
Eviction with Reduced EvSet	Section 6.2
EvSet Const	Section 8.1.1
Reverse Engineering of DDIO	Section 5

The first experiment is provided as a warm-up to the basic API usage, the building of the attacker-victim framework, eviction set creation, and cache-timing measurements. For each experiment, a dedicated documentation file is provided. This file explains the conducted experiment, how to execute it, and the expected results.

G.3.1 How to access

The artifacts are published on GitHub. The version, improved with evaluators' suggestions, is tagged as 'ArtifactsAvailable'.

The Stable URL is: https://github.com/KULeuven-COSIC/Double-Trouble/tree/ArtifactsAvailable

We wish to keep the repository solely for the artifacts of the paper, so the most recent commit should always apply for this paper.

G.3.2 Hardware dependencies

Our work requires an FPGA acceleration card. Specifically, we tested with the following Intel PACs (Programmable Accelerator Cards):

- Intel Arria 10 PAC
- Intel Stratix 10 PAC

One can either have a local setup that employs an Intel PAC or work with remote access to such a platform. We did both. For the latter, Intel Labs (IL) Academic Compute Environment (ACE)

(https://wiki.intel-research.net/FPGA.html) provides various remotely accessible platforms.

For people who do not have access to an FPGA accelerated platform but are still interested in our findings, we provided detailed documentation on our repository. The documentation includes a separate file for each experiment, where we provide an example output of the execution for proving the claims we made out of them.

G.3.3 Software dependencies

On a platform with Intel PAC, we can assume that the corresponding Intel OPAE SDK (https://github.com/OPAE) is available.

The compilation is straightforward with `Makefile`s. We tested on two different setups with different `gcc` versions:

- On our local setup: Ubuntu 18.04.5 LTS with `gcc` 7.5.0
- On Intel Labs ACE: CentOS Linux 7.7.1908 with `gcc` 4.8.5

A few experiments require installing additional libraries. Instructions to install them are available for each repository, though they need root privileges on the machine.

- `intel-cmt-cat` https://github.com/intel/intel-cmt-cat is used for the experiments in Section 5. This library allows to fix the LLC ways used by a CPU core.
- `intel-msr-tools` https://github.com/intel/msr-tools is used for changing the default cache configuration, with the purpose of giving the FPGA (or DDIO in general) broader cache access. Experimenting with these non-default configurations was a suggestion by the reviewers.

G.3.4 Security, privacy, and ethical concerns

Our work demonstrates a timing-based cache side-channel framework, aiming for the disclosure of our security and privacy concerns associated with the underlying platforms.

G.4 Installation

Cloning the repository is adequate for obtaining the source files.

G.5 Experiment workflow

All experiments consist of two steps; compilation with a provided Makefile and execution of the generated binary. Depending on the platform, there can be various customizations, e.g., pinning processes to specific CPU cores.

Each experiment in the repository comes with a separate documentation file. These documents provide experiment-specific compilation (with `Makefile` targets) and execution commands, besides the experiment explanation and expected outputs.

G.6 Evaluation and expected results

The evaluation is divided into multiple experiments. The initial experiments help understand the non-inclusive LLC structure and its DDIO-based access. The later experiments entail:

- reverse engineering the underlying mechanisms,
- demonstrating the findings, and
- analyzing the performance.

G.6.1 Figure 4

We have four experiments respectively for the observations provided in Figures 4c, 4d, 4e and 4f. The expected results are timing measurements that support the claimed observations in Figures 4 and Section 3.2.

G.6.2 Cache Timing Histogram

This experiment measures the timings for cache line accesses from various levels in the cache hierarchy and constructs a histogram that helps to distinguish accesses by their latency. The expected result is a histogram similar to Figure 9 given in Appendix A, though the timings can vary on different platforms.

G.6.3 Eviction Candiate

This experiment determines whether DDIO reads and writes are recorded by the cache replacement policy, i.e., whether they change the eviction candidate. The expected results consist of cache timing measurements prooving the claims made in Section 3.2.2 including Figure 3.

G.6.4 DDIO Replacement Policy

This experiment performs various access patterns with DDIO lines, checks the cache contents after these accesses, and compares them with the expected contents for different replacement policies. The expected result is the re-construction of Table 10.

G.6.5 Reduced Eviction

This experiment implements the eviction with the reduced-eviction approach explained in Section 6.2. It creates random bits, indicating whether the victim accesses the target address or not. The attacker monitors the victim's activity and determines whether the victim has accessed the target. The expected results are the measures indicating the success of eviction with a reduced eviction set.

G.6.6 Eviction Set Construction

This experiment is used to evaluate the eviction set construction performance for various configurations, which are:

- Non-default DDIO way settings
- Huge or small pages
- Different levels of stress
- Options of congruence checks integrated into the eviction set construction

The expected results comprise a debug log of FPGAs eviction set construction e.g., how many guesses were needed for every congruent address, total construction time, and failed attempts. These results are used in Section 8.1.1 and to construct Figure 8.

G.6.7 Reverse Engineering the DDIO and DDIO+ regions

This experiment is used to reproduce the findings in Section 5. The expected results indicate the ways available to DDIO and DDIO$^+$ allocations, hence allowing to redraw Figure 7.

G.7 Experiment customization

The experiments offer limited customizability and interactions, including picking the associations of processes to specific CPUs.

Specifically for the *Eviction Set Construction* experiment, various measurements, indicated in Appendix G.6.6, can be perfomed by customizing the command line arguments to the binary.

G.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact goes through the 3 main steps of the evaluation with FixReverter and RevBugBench: (1) FixReverter bug injection, (2) FuzzBench experiments, and (3) FixReverter bug triage. The evaluation results of 5 different fuzzers on a benchmark generated by FixReverter, namely the RevBugBench, show that FixReverter is able to generate hard-to-find bugs and differentiate the performance of fuzzers. As the full-scale experiments require a lot of time and resources, the artifact provides all the intermediate products of each step for partial reproductions. A machine with Ubuntu system, at least 24 CPU cores and 200GB RAM is recommended for the experiments.

A.2 Artifact check-list (meta-information)

- **Run-time environment:**

  Ubuntu 16.04, Docker 20.10.7 and python 3.9.

- **Hardware:**

  At least 200GB RAM and a 24-core CPU are recommended.

- **Output:**

  Performance of 5 fuzzers on RevBugBench.

- **How much disk space required (approximately)?:**

  500GB if running the full-scale evaluation. This can be reduced by running only partial experiments.

- **How much time is needed to prepare workflow (approximately)?:**

  2 hours.

- **How much time is needed to complete experiments (approximately)?:**

  One week if running the full-scale evaluation. Running partial experiments with provided intermediate products can take from 1 hour to several days.

A.3 Description

A.3.1 How to access

figshare URL: `https://figshare.com/articles/software/Supplementary_artifact_for_the_paper_FIXREVERTER_A_Realistic_Bug_Injection_Methodology_for_Benchmarking_Fuzz_Testing_/20647821`

DOI: 10.6084/m9.figshare.20647821

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

Ubuntu 16.04, Docker 20.10.7 and python 3.9. Other dependencies (Clang, FuzzBench and Phasar) are automatically handled in the provided docker images.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Installation guides are included in the README of the artifact.

A.5 Experiment workflow

The workflow is described in detail with the README of the artifact. Each section comes with numbered steps explaining the workflow, and necessary actions come with highlighted commands.

A.6 Evaluation and expected results

We made 3 major claims in the evaluation of the paper.

- FixReverter injects bugs that fuzzers can actually find.
- FixReverter injects bugs that are hard to find.

- Fuzzers can find combination causes in RevBug-Bench.

First, the results show hundreds of bugs can be found by the 5 evaluated fuzzers. Second, some observations of the difference in fuzzers' performance shows the difficulty for fuzzers to find the injected bugs, as described in Section 5.2 of the paper. For example, each fuzzer detected unique bugs that other fuzzers did not find, indicating that injected bugs do not overfit a single approach in the evaluated fuzzers. Finally, there are hundreds of combined causes identified in the results. Because fuzzing is a random process, this artifact is expected to produce results that support the above 3 claims and are reasonably similar to the numbers reported in Section 5.

A.7 Experiment customization

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

This artifact description contains information about a prototype implementation of PRIVANALYZER. The implementation is composed of (1) a policy parser; (2) a static analyzer; and (3) a set of function summaries. Code for performance evaluation on Parcel is not provided due to non-disclosure concern.

A.2 Artifact check-list (meta-information)

- **Algorithm: Parse policy strings in LEGALEASE to disjunctive normal form.**
- **Program: LEGALEASE parser + PRIVANALYZER + a set of function summaries.**
- **Run-time environment: Ubuntu 16.04 LTS.**
- **Execution: See below.**
- **Output: Residual policy.**
- **Experiments: See below.**
- **Publicly available?: Code and example test cases are publicly available.**
- **Code licenses (if publicly available)?: MIT License.**

A.3 Description

A.3.1 How to access

The codebase can be accessed from Github https://github.com/sunblaze-ucb/privguard-artifact with commit hash b1b5f3a16af6ab5f7cb0f0737aba27dd9d76c25b. We are still actively updating the codebase. To access the newest version, please use https://github.com/sunblaze-ucb/privguard-artifact.

A.3.2 Software dependencies

To run PRIVANALYZER, python3.6 and python3.6-venv are required. Additional Python package dependencies are as follow.

- pypandoc==1.6.4
- pyparsing==3.0.0rc2
- numpy==1.19.5

A.4 Installation

The statis analyzer has been tested in Ubuntu 16.04 system. To run the static analyzer, pleaes install python3.6 and python3.6-venv using the following lines.

```
sudo apt install python3.6
sudo apt install python3.6-venv
```

To download our codebase, run

```
git clone https://github.com/sunblaze-ucb/
    privguard-artifact.git
```

Then create and activate a python3.6 virtual environment, install python packages, and set environment variables by running

```
chmod u+x path-to-repo/setup.sh
path-to-repo/setup.sh
```

A.5 Evaluation and expected results

In this codebase, we provide test scripts for the policy parser and the static analyzer separately.

Policy Parser. To test the policy parser, run

```
python path-to-repo/src/parser/policy_parser
    .py
```

and input a valid policy string (e.g. "ALLOW FILTER age >= 18 AND SCHEMA NotPHI, h2 AND FILTER gender == 'M' ALLOW (FILTER gender == 'M' OR (FILTER gender == 'F' AND SCHEMA PHI))") in Legalease. The program will output the policy translated to Python objects.

To test converting a policy into its DNF form, run

```
python path-to-repo/src/parser/policy_tree.
    py
```

Static Analyzer. We provide 5 example programs to test the static analyzer. To run these examples, use the following script with correct flag values. Please make sure your environment variable is correctly set before testing the below functionality (see setup.sh for more information).

```
python path-to-repo/src/analyze.py --
    example_id XX
```

We provide details about two examples below.

ELECTRICAL HEALTH RECORD (0): The first example loads two data files: patients/data.csv and conditions/data.csv whose policies are as below.

```
ConjunctClause(redact: NAME(None:None),
    filter: AGE [e18, einf])
ConjunctClause(filter: CONSENT [eY, eY],
    filter: DESCRIPTION [
    eViralSinusitisDisorder,
    eViralSinusitisDisorder], privacy:
    Aggregation)
```

The residual policy is

```
ConjunctClause(UNSAT)
```

which means the policy is unsatisfiable. The reason is that in the last line of the program, the DataFrame calls its groupby method which prevents any further operation to satisfy the `PRIVACY Aggregation` attribute.

TRANSACTION PREDICTION (6): The second example loads one data file: train/data.csv whose policy is as below.

```
ConjunctClause(privacy: Aggregation, redact:
    ID(None:None))
```

The program drops the ID column and trains a model on the data, so the guard policy is fully satisfied and the residual policy is

```
ConjunctClause(SAT)
```



A Artifact Appendix

A.1 Abstract

We developed OVRSEEN, a methodology and system for collecting, and analyzing network traffic and privacy policies on OVR. OVRSEEN consists of two main parts: network traffic and privacy policy. OVRSEEN's network traffic part consists of traffic collection and post-processing. First, traffic collection consists of repackaging the app's APK, and running the traffic decryption scripts simultaneously with AntMonitor, the traffic collection app. Second, post-processing consists of scripts that perform analysis on the collected network traffic.

OVRSEEN's privacy policy part consists of network-to-policy consistency analysis, and purpose extraction. First, the network-to-policy consistency analysis comes in the form of PoliCheck that has been adapted with VR data and entity ontologies to perform its analysis on VR apps. Second, the purpose extraction comes in the form of scripts that send privacy policies to Polisis website (https://www.pribot.org/) using the provided REST APIs, and scripts that provide a translation from PoliCheck data flows into text segments that have been annotated by Polisis with purposes.

A.2 Artifact check-list (meta-information)

- **Datasets:** lists of apps, network traffic dataset, privacy policy files, manual validation of PoliCheck and Polisis spreadsheets, and intermediate output files.
- **Run-time environment:** Python scripts tested on Python 3.8 and 3.9; we also provide a VM that runs Ubuntu 20.04.3 LTS with all the dependencies installed.
- **Hardware:** an Oculus Quest 2 device and a standard machine with Linux/MacOS (or using the provided VM).
- **Execution:** We have provided a set of steps to demo OVRseen (*e.g.,* for artifact evaluation), which takes a few hours approximately.
- **Metrics:** number of packets, TCP flows, numbers of apps, domain names/entities, and data flows.
- **Output:** console output (*e.g.,* debug/error messages), intermediate output files, final analysis results.
- **Experiments:** Please see OVRSEEN's Github page (*i.e.,* "Try OVRSEEN Yourself" Wiki page in particular).
- **How much disk space required (approximately)?:** Our VM provides 30GB of disk space (more than 25GB will be occupied when running OVRSEEN).
- **How much time is needed to prepare workflow (approximately)?:** Downloading and booting up the provided VM should take less than 1 hour (plus additional time to install VirtualBox/VMWare).
- **How much time is needed to complete experiments (approximately)?:** We have a quick demo for OVRSEEN that would take a few hours (at least 2 hours).
- **Publicly available?:** We have released OVRSEEN, along with the datasets, publicly.
- **Code licenses (if publicly available)?:** Code licenses information is available on OVRSEEN's Github page.
- **Data licenses (if publicly available)?:** Datasets licenses information is available on the datasets release page.
- **Archived (provide DOI or stable reference)?:** https://doi.org/10.5281/zenodo.5565170

A.3 Description

A.3.1 How to access

We have made OVRSEEN available at https://athinagroup.eng.uci.edu/projects/ovrseen/ and our datasets at https://athinagroup.eng.uci.edu/projects/ovrseen-datasets/. These two links have the complete information about the paper, OVRSEEN, and datasets. For convenience, the link to OVRSEEN's Github page is https://github.com/UCI-Networking-Group/OVRseen.

A.3.2 Hardware dependencies

OVRSEEN's network traffic and privacy policy analyses can be run on our datasets on a standard machine that runs Linux/MacOS, or using the provided VM. The information to download the VM can be found at https://github.com/UCI-Networking-Group/OVRseen#getting-started. Please see the "Virtual Machine" section: https://github.com/UCI-Networking-Group/OVRseen#virtual-machine.

OVRSEEN's network traffic collection needs a real Oculus Quest 2 device for the most part. Thus, our quick demo mainly assumes that one just runs OVRSEEN on our datasets (without collecting network traffic on the device).

A.3.3 Software dependencies

The dependencies for OVRSEEN are explained in detail on OVRSEEN's Github Wiki page at https://github.com/UCI-Networking-Group/OVRseen/wiki#dependencies. These dependencies have been properly installed and set up in the provided VM.

A.3.4 Datasets

OVRSEEN has a number of datasets: (1) list of apps in our corpus (*i.e.,* two files that contain apps information obtained by crawling the Oculus and SideQuest app stores, and four files that contain the information of the top 150 apps); (2) network traffic dataset in the form of PCAP files from 140 VR apps; (3) 102 privacy policy files; (4) manual validation results for PoliCheck and Polisis (*i.e.,* two spreadsheets); and (5) intermediate outputs (*i.e.,* a CSV file containing TCP flows, pre-processed privacy policy files, a CSV file containing data flows, PoliCheck output files, JSON files containing Polisis output for text segment annotation, and a CSV file that contains the translation/mapping from PoliCheck data flows into the annotated text segments from Polisis). For artifact evaluation purposes, we provide the download link for our datasets through hotcrp.com. In the future, these datasets will be shared

to OVRSEEN users after they submit the consent form at https://athinagroup.eng.uci.edu/projects/ovrseen-datasets/.

A.3.5 Security, privacy, and ethical concerns

Please keep in mind that VR apps collect PII and other sensitive information: OVRSEEN collects such sensitive information as well when used to collect and analyze network traffic. Our network traffic dataset, however, contains PII that is associated only with a test account/persona (*i.e.,* no human subjects were involved).

A.4 Installation

We have provided complete instructions on how to download, install, and use OVRSEEN on its Github page: https://github.com/UCI-Networking-Group/OVRseen (please see the README and Wiki pages). The instructions also include how to download and use our VM that has all the dependencies installed.

A.5 Experiment workflow

We have created a Wiki page (called "Try OVRSEEN Yourself") on OVRSEEN's Github page (*i.e.,* https://github.com/UCI-Networking-Group/OVRseen/wiki/Try-OVRseen-Yourself). This Wiki page contains a set of steps that one can follow to quickly demo OVRSEEN's workflow using our datasets.

A.6 Evaluation and expected results

Main claims. Our paper presents OVRSEEN, a methodology and system for collecting, and analyzing network traffic and privacy policies on OVR. In our paper, we first claimed that, using OVRSEEN, we decrypted, captured, and analyzed network traffic of VR apps. Then, we made the following claims based on our findings:

- *More centralized, more tracking, but less advertising*: the OVR ecosystem is more centralized, and driven by tracking and analytics, instead of by third-party advertising.
- *Data types exposure*: data types exposed by VR apps include the traditional PII and, most notably, VR specific data types.
- *Inconsistent disclosures*: the majority of data type exposures of an app are inconsistent with the disclosures in the app's privacy policy.
- *Non-core purposes*: many data exposures occurred for purposes unrelated to an app's core functionality.

Key results. Next, we outline the key results that support our main claims:

- *More centralized, more tracking, but less advertising*: We found that OVR exposes data primarily to tracking and analytics services, and has a less diverse tracking ecosystem. We found no evidence of data exposure to advertising services as ads on OVR is still in its infancy (see Section 3.3).
- *Data types exposure*: We discovered that there were 21 data types exposed, namely *PII*, *Fingerprint*, and *VR Sensory Data* data types (see Section 3.4).
- *Inconsistent disclosures*: First, we found that approximately 70% of data flows from VR apps were inconsistent with their privacy policies: only 30% were consistent. Second, apps' privacy policies often neglected declaring privacy policies from the libraries they used. We discovered that by including these other parties' privacy policies in OVRSEEN's network-to-policy consistency analysis, 74% of data flows became consistent (see Section 4.1).
- *Non-core purposes*: We discovered that there were 69% of data flows that have purposes unrelated to the core functionality, *e.g.,* advertising, marketing campaigns, and analytics (see Section 4.2).

Reproducing key results. To reproduce the key results, we recommend our artifact reviewers to follow the instructions at https://github.com/UCI-Networking-Group/OVRseen/wiki/Try-OVRseen-Yourself that we also describe in detail in the following.

To prepare OVRSEEN, please follow the instructions in the "Virtual Machine" section in the README (*i.e.,* https://github.com/UCI-Networking-Group/OVRseen#virtual-machine) to first download and run our pre-configured VM (with all the dependencies installed). Then, our reviewers can download, install (*e.g.,* in the home directory), and run OVRSEEN on the running VM.

First, collecting network traffic using OVRSEEN is not possible without installing AntMonitor and running the certificate validation bypass scripts on a real Oculus Quest 2 device. Further, it is impractical to repeat our network traffic collection steps on 140 VR apps for the purpose of artifact evaluation. Thus, we release our network traffic dataset in the form of PCAP files that we captured using AntMonitor and the certificate validation bypass scripts. We welcome our reviewers to download and use our datasets and run OVRSEEN on them: this will be sufficient to reproduce all results we reported in our paper.

Since OVRSEEN's *traffic collection* is impractical to perform for our reviewers, we invite our reviewers to look at the complete source code for AntMonitor and the certificate validation bypass scripts. We also invite our reviewers to look at https://github.com/UCI-Networking-Group/OVRseen/wiki/Traffic-Collection to review the instructions: these have been tested using our VM and a real Quest 2 device. One part of the OVRSEEN's *traffic collection* that our reviewers can still run in the quick demo is the app repackaging step—we provide a sample app to test with (please see https://github.com/UCI-Networking-Group/OVRseen/wiki/Try-OVRseen-Yourself#traffic-collection).

Next, using the provided network traffic dataset (and our other datasets), our reviewers can perform the following steps when running OVRSEEN.

- *More centralized, more tracking, but less advertising*: OVRSEEN's *post-processing* scripts can be run to analyze the network traffic dataset we provide; the final product of OVRSEEN's *post-processing* is a combined CSV file that contains information on TCP flows: each TCP flow, among other information, records app ID (*i.e.,* app name), PII types, and endpoints; for now, we recommend that OVRSEEN is run partially on our network traffic dataset (due to the limitations of RAM and disk space in the VM), but we provide the intermediate outputs generated when we ran OVRSEEN on our complete

network traffic dataset and scripts that use these outputs to reproduce Table 1 (discussed in Section 3.2.1), and Table 2 and Figure 2 (discussed in Section 3.3) in our paper.

- *Data types exposure*: OVRSEEN's *post-processing* also includes scripts that use the intermediate outputs to reproduce Table 3 that summarizes data types exposures, destinations, and blocklists' effectiveness for 21 data types; we discuss these in Section 3.4 in our paper.
- *Inconsistent disclosures*: OVRSEEN's *network-to-policy consistency* analysis consists of PoliCheck that has been adapted and improved for VR apps, and VR (data and entity) ontologies; our reviewers can run OVRSEEN's *network-to-policy consistency* analysis using the provided intermediate outputs to reproduce our results reported in Section 4 in our paper: more specifically, we provide scripts to reproduce Figures 4, 5, and 6; further, we release the HTML files of the 102 privacy policies we collected, the scripts that pre-process these into text files suitable as an input to PoliCheck, and the text files themselves; in addition to privacy policies, PoliCheck also takes a CSV file that contains data flows information extracted from the CSV file that contains TCP flows information (*i.e.,* output of OVRSEEN's *post-processing*): we release the scripts to produce this data flows CSV file along with the CSV file itself; finally, we also provide the output CSV files from PoliCheck's disclosure classification and the spreadsheet that contains the results of our manual validation for PoliCheck.
- *Non-core purposes*: OVRSEEN's *purpose extraction* consists of scripts that perform the extraction of purposes for text segments in privacy policies using Polisis, and scripts that perform translation/mapping from PoliCheck data flows to the text segments annotated by Polisis; Polisis website requires a special token for the APIs to work with our scripts; unfortunately, while the token can be acquired by contacting Polisis authors, they had to discontinue their online service as of September 2021 due to some technical issue; thus, we provide the JSON files that contain the Polisis analysis output we obtained for our 102 privacy policies; using these files, one can run our scripts to reproduce the statistics/results we reported in Section 4.2 and Figure 7 in our paper; further, we also release the spreadsheet that contains the results of our manual validation for Polisis.

Thus, we believe that the instructions we provide at https://github.com/UCI-Networking-Group/OVRseen/wiki/Try-OVRseen-Yourself are sufficient to quickly demo OVRSEEN. While, parts of OVRSEEN's workflow will not be possible to perform (*e.g.,* the network traffic collection that requires a Quest 2 device, the Polisis online service that has been discontinued, *etc.*), these instructions, coupled with our datasets, will allow our artifact reviewers to reproduce our (key) results to support the main claims in the paper.

A.7 Experiment customization

If one has a local machine that allows the provided VM to be provisioned with more RAM and disk space, they can try to increase the RAM and disk space for the VM, and run OVRSEEN on our entire datasets. Please see https://github.com/UCI-Networking-Group/OVRseen/wiki/Try-OVRseen-Yourself for more information.

Further, OVRSEEN can be used to collect network traffic from other apps on Quest 2. The PCAP files can then be post-processed and analyzed (together with the apps' privacy policies) using OVRSEEN. For other devices, other than Quest 2 (or even non-VR devices), one has to adapt the network traffic collection part to decrypt network traffic on the device. Other parts of OVRSEEN also may need adjustments if the collected network traffic contains new data types. For instance, new network traffic dataset and/or privacy policies may change PoliCheck's data and entity ontologies.

Additionally, we also release our app crawler scripts that we used to collect app information we present in our lists of apps, and the curated lists of top apps. Please see https://github.com/UCI-Networking-Group/OVRseen/wiki/App-Corpus for more information on how to use them. Nevertheless, we do not consider these crawler scripts to be part of the main OVRSEEN's workflow.



A Artifact Appendix

A.1 Abstract

This paper presents Half-Double, a new Rowhammer effect extending the reach of Rowhammer beyond the immediate neighbors. We show that this effect can not only circumvent current state-of-the-art mitigations like TRR, but defensive refreshes to *distance-1* rows also assist Half-Double. The general idea is to induce flips into a victim by combining many *distance-2* accesses with a few *distance-1* accesses.

In the artifact evaluation, we present experiments to underline the impact of Half-Double. Due to obligatory constraints, we cannot share parts of the initial root-cause analysis. Nevertheless, the artifacts presented show all the necessary steps to mount the Half-Double Attack on commodity systems protected by TRR and ECC.

We split the artifacts into the described challenges, which finally form the end-to-end exploit. First, the artifacts for Challenge **C1** "Memory Allocation" demonstrate three different ways to reconstruct contiguous memory. Second, for Challenge **C2** "Alternatives to Memory Templating", we show both ECC-aware hammering and Blind-Hammering and provide the utility to count the overall bitflips on a device. Third, Challenge **C3** "Memory Preparation" shows the *Child Spray* technique to fill the memory with attackable data, i.e., page tables. Fourth, we provide the artifacts for **C4** "Robust Bit-Flip Verification", namely the speculative oracle and the architectural `vfork` alternative. Finally, the Half-Double Attack built upon the previous parts to mount the end-to-end attack.

The end-to-end exploit is optimized for the chromeOS operating system and, more precisely, for our Chromebook setup. Nevertheless, all the components are compileable for both x86 and aarch64 architectures. We recommend ARM-v8 and Intel x86 CPUs for this artifact evaluation.

A.2 Artifact check-list (meta-information)

- **Program:** We provide the programs and represent how to install them.
- **Compilation:** We require gcc for cross-compilation. Download instructions are provided.
- **Run-time environment:** We require a native Linux installation for compilation. Some artifacts can be directly executed under Linux. For this purpose, we strongly recommend Ubuntu 20.04. For the end-to-end exploit, we require a chromeOS installation. The provided installation instructions need internet access.
- **Hardware:** We require either Intel x86 CPUs or ARM-v8 CPUs. Half-Double bitflips depend highly on the actual hardware and even differ between identical DRAM modules.
- **Execution:** For executing some benchmarks, we require a stable frequency.
- **Security, privacy, and ethical concerns:** Due to the Half-Double bitflip effect, **data corruption** can occur on the used system.
- **Metrics:** The benchmarks report nanosecond execution time, data size in bytes, and throughput in mega- or gigabytes per second.
- **Output:** The artifacts print the results to the terminal.
- **Experiments:** We include the source code, build scripts, and readmes describing the artifact and the process of how to execute the benchmarks.
- **How much disk space required (approximately)?:** Less than 1 GB.
- **How much time is needed to prepare workflow (approximately)?:** Below 4 hours.
- **How much time is needed to complete experiments (approximately)?:** Up to two days, depending on the hardware.
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/iaik/halfdouble
- **Code licenses (if publicly available)?: MIT**
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/iaik/halfdouble/tree/ae

A.3 Description

A.3.1 How to access

Check out the Git repository from https://github.com/iaik/halfdouble and follow the provided readmes.

A.3.2 Hardware dependencies

We recommend ARM-v8 CPUs with (LP)DDR4(x) DRAM supporting both TRR and ECC, like the Chromebooks in the paper. Most of the artifacts can also be executed on Intel x86 CPUs. Our experience showed that the susceptibility to Half-Double is highly dependent on the used DRAM modules.

A.3.3 Software dependencies

We strongly recommend Ubuntu 20.04 as a platform for compilation as we tested all the building steps there. The operating system to execute the artifacts should either be an Ubuntu or chromeOS operating system with root access for debugging. The components of the paper have to be built from

the source. Hence the system requires tools for compiling software (`build-essentials` on Ubuntu). Finally, access to operating system interfaces as root is necessary for debugging, e.g., `/proc/self/pagemap` and `/dev/mem`.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

During our experiments with Half-Double, we observed **data corruption** in the operating system resulting in corrupted file systems. Therefore, we highly recommend a fresh installation with an operating system image not used for personal or important data. We *never* observed persistent damage on the hardware. However, we cannot ensure this is generally the case, but we find it highly unlikely to damage the used hardware.

A.4 Installation

Follow the readmes in the repository's top-level directory, which will guide you through installing all the necessary tools and components of the paper. The "Makefiles" *should* automate most of the process. However, we cannot rule out that some parts might need manual adjusting, and therefore, knowledge of C, C++, python3, bash, and Makefiles is beneficial.

A.5 Experiment workflow

Each artifact contains a readme, the source code, and a build script to build the source. After the binary is compiled, we can reuse the build script to *deploy* the binary to the test systems where the binary is executed. Note that some binaries require additional arguments passed via the terminal. The binary prints debug output to the terminal, and the results are also reported in this way.

A.6 Evaluation and expected results

The evaluation is split into multiple parts. First, we use the provided Half-Double hammering tool to verify the results from Table 1. The tool uses the *Quad pattern* to hammer and induce flips on commodity devices, e.g., the provided Chromebooks. The tool should report similar flip frequencies if performed on the provided hardware. Second, we execute the artifacts of Challenge **C1** to verify the general functionality and the performance numbers of Section 6.1 when detecting contiguous memory. Third, for Challenge **C2** we reuse the hammering tool with a slightly different configuration to demonstrate both Blind-Hammering and ECC-aware templating from Section 6.2. Fourth, Challenge **C3** uses an executable to demonstrate the *Child spray* of Section 6.3 to circumvent some ARM CPUs' reduced virtual address space and verify the performance numbers. Finally, the artifacts of Challenge **C4** scan memory and test the bitflip verification of Section 6.4 if a page table is corrupted.

A.7 Experiment customization

The artifacts use a timing side channel to find addresses belonging to the same DRAM bank. Therefore, the threshold of the timing side channel is configurable and usually passed via a command-line argument. We provide an additional utility to evaluate this threshold empirically. Nevertheless, this threshold might need manual adjustment. Finally, we can adjust the number of repetitions of a benchmark and the performed accesses in the hammer loop via compile-time parameters.

A.8 Notes

Rowhammer bitflips depend highly on the used DRAM, the device's battery state, and the environment. Similar to Table 1, identical commodity systems can behave differently. Therefore it is likely that results from the artifacts may differ.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

Using RETBleed, an unprivileged user can leak arbitrary memory from the system. The vulnerable systems are listed in Table 1 in the paper. RETBleed has an offline phase, where exploitation primitives are discovered by our framework. We ran the framework, designed exploits and proof of concept code on Ubuntu 5.8.0-63-generic. To match our results closely, an Intel i7-8700K (Coffee Lake) and an AMD EPYC 7252 (Zen2) are recommended. 16 GiB of RAM is recommended. To run our framework on the entire test suite (optional), we recommend 40 GiB of free disk space. However, we have included example output from the framework with particularly huge files omitted, which is 1-2 GiB and can be inspected instead.

We provide a snapshot of the current RETBleed repository, which hosts the majority of code used throughout the project. The repository is public, and detailed instructions are found in the `README.md` files inside the git repository.

A.2 Artifact check-list (meta-information)

- **Binary:** A Linux image is included to test the gadget scanner. Source code is included to build the other binaries used by the framework, PoCs, kernel modules and end-to-end exploits.
- **Run-time environment:** Ubuntu 20.04.3 LTS (Focal Fossa) with `linux-image-5.8.0-63-generic`. Most experiments are designed to run on bare-metal, not on a VM.
- **Hardware:** Intel Core generations 6–8; AMD Zen, Zen+ and Zen 2
- **Security, privacy, and ethical concerns:** Responsible disclosure ended on 12 July 2022.
- **Experiments:**
  - `./retbleed_zen/pocs/ret_bti` finds the patterns that cause BTB collisions.
  - `./retbleed_zen/pocs/cp_bti` shows that collisions happen across.
  - `./retbleed_intel/pocs/ret_bti` shows that returns go via BTB.
  - `./retbleed_intel/pocs/cp_bti` shows that we can train across kernel returns in user space.
  - `./rsb_depth_check` RSB use on AMD and Intel. For Intel, it also indicates that some other "near branch" prediction mechanism takes place.
  - `./zen_ras_vs_btb/` is illustrated in Figure 5. It shows that Return Address Stack (RAS, aka RSB) is not used on Zen 2 when there's a BTB entry. To evaluate Zen(+), `BTI_PATTERN` must be manually set.
  - `./ret_finder/` constitutes the part of framework to detect vulnerable return instructions in the kernel.
  - `./gadget_scanner/` was used to discover disclosure gadgets.
  - `./bhb_generate/` was used to trace taken branches preceding a vulnerable return in a kernel running inside a VM.
- **How much disk space required (approximately)?:** 300 MiB. 40 GiB to reproduce our framework output.
- **How much time is needed to prepare workflow (approximately)?:** Less than 1 hour.
- **How much time is needed to complete experiments (approximately)?:** Up to 12 hours.
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/comsec-group/retbleed/.
- **Workflow frameworks used?:** git, linux-test-project, BCC/eBPF, ftrace
- **Archived:** https://github.com/comsec-group/retbleed/releases/tag/sec22-artifact-final

A.3 Description

A.3.1 How to access

Clone using git, `git clone https://github.com/comsec-group/retbleed.git`. Also clone submodules, `git submodule update -init`

A.3.2 Hardware dependencies

Intel i7-8700K (Coffee Lake) and AMD EPYC 7252 (Zen 2) or similar. We were evaluating all experiments and exploits on bare-metal hardware. Running in a VM may pose unexpected challenges.

A.3.3 Software dependencies

Ubuntu focal, linux-image-5.8.0-63-generic, clang, python3, bcc, bpfcc-tools, pytest, pyelftools

A.4 Installation

Instructions available in *README.md* files. See repository for details. Software dependencies can be installed using `apt-get` and `pip3`. Linux test project included as a submodule that can be cloned using `git submodule update -init` from the repository

A.5 Experiment workflow

Instructions available in *README.md* files. See repository.

A.6 Evaluation and expected results

- **Reverse engineering of return instruction behavior.** Several experiments are included that reverse engineer return behavior.
- **Framework that finds vulnerable return instructions.** We include the framework that finds these. It should result in the numbers from Figure 11.

- **Poisoning kernel returns from an unprivileged process.** Our PoCs and exploits all do this.
- **Leaking arbitrary memory at 3.9 kB/s and 219 bytes/s on AMD Zen2 and Intel Coffee Lake respectively.** We provide instructions in the repository for how to run these PoCs. We also include exploits to leak */etc/shadow*. Furthermore, we also explain how we measure the leakage rate. The median the leakage rate should closely match with the expected results.

A.7 Experiment customization

We clarify in the READMEs provided the cases where certain preprocessor macros can/should be altered for additional results. For example, to run `rsb_depth_check` on AMD, uncomment L11 in `ret_chain.c`.

A.8 Notes

The documentation here is sparse, since everything written here has already been provided in the artifact project itself. Please use your own hardware. Should you not have access to hardware that is similar to ours, please contact us.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artefact Appendix

A.1 Abstract

Pistis artefact is a set of source files and scripts that can be evaluated partially on a standard Linux environment (local evaluation), and partially with the support of a MSP430F5529LP micro controller unit (MCU). However, we provide the reviewers with an SSH access to a VM connected to one of such board (remote evaluation). The VM is shared between the reviewers and does not allow multiple users to interact with the MCU at the same time. The reviewers are thus asked for their collaboration in sharing such VM instance. For the local evaluation, the reviewers will be asked to compile the core of Pistis and use the available scripts to compile the user-applications. For the remote evaluation, using the VM, they will be asked to check Pistis at runtime, debugging its execution using a GUI-based IDE.

A.2 artefact check-list (meta-information)

- **Program:** TI MSP430 Benchmark, custom test bench
- **Compilation:** msp430-gcc-9.2.0.50, included, public
- **Transformations:** python-script
- **Run-time environment:** Linux, non-root
- **Hardware:** x86_64 Machine, (optional) MSP430F5529LP
- **Output:** console, graphical, interactive
- **Experiments:** Python scripts, bash scripts, CodeComposerStudio (CCS), debugging
- **How much disk space required (approximately)?:** 10GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 2 hours
- **Publicly available (explicitly provide evolving version reference)?:** `https://github.com/MicheleGrisafi/PISTIS_AE`
- **Code licenses (if publicly available)?:** The 3-Clause BSD License
- **Archived (explicitly provide DOI or stable reference)?:** `https://github.com/MicheleGrisafi/PISTIS_AE/releases/tag/Artefact.v1`

A.3 Description

Pistis is a Trusted Execution Environment (TEE) developed for MSP4305529 Micro Controller Units (MCUs). Being a fully software based TEE, its features are many spanning a complex software structure. Although we provide the entirety of Pistis, with all of its modules and test applications (as presented in the papers), we only provide instructions on how to evaluate part of it. This is due to the complex and time-demanding nature of a complete evaluation, which would also require a MSP4305529LP MCU.

In particular, we provide instructions on how to: (i) build the core of Pistis, (ii) use the custom toolchain to compile applications, (iii) check the run-time verification process, (iv) check the run-time memory protection. While some of these operations can be performed with a local environment (based on Ubuntu 20.04), others require SSH access to a shared VM connected to a MSP4305529LP board. This access must be shared between any reviewer who cannot interact concurrently with the board. The local evaluation will only require to run a few CLI commands, while the remote evaluation will require the reviewers to operate on an Eclipse-based IDE and its debugger. To facilitate the operations, we provide a few video tutorials.

A.3.1 How to access

The artefact can be downloaded from the official artefact github repository: `https://github.com/MicheleGrisafi/PISTIS_AE`. It is worth noting that this repository is based on the official repository (link in the official paper). Given the strict hardware requirements for the evaluation of this artefact, we prepared a virtual machine (with ssh access) connected to a single MCU, the one we used in our experiments. Although this setting poses some limitations to the reviewers, it can be used to have a deeper inspection on how Pistis functions.

In order to access the VM we set up, the following command should be executed on a local graphical-based Linux environment: *no more available*[1]. This will establish a two-hops ssh connection to the private VM passing through a public VPS. The passwords for the two hops, which will be required upon each connection, are the following: `Pistis1940` for the pistisAE user (first hop) and `pistis` for the pistis user (the VM). The `-Y` option enables the forwarding of the graphical environment, thus allowing the reviewers to visualise on their own machine any GUI lunched on the remote machine.

A.3.2 Hardware dependencies

In order to compile our binaries, an x86_64 Linux based machine should be used. Optionally, a MSP430F5529LP MCU can be used to perform locally also the remote evaluation. Nevertheless, this artefact provides the reviewer with a single shared VM instance connected to one of such MCUs. This will help the reviewers to execute code on the board. As a consequence, we proceed to describe two environment: a local environment (**local**) for the MCU-independent tests, and an MCU-connected environment (**remote**) for all the other tests.

A.3.3 Software dependencies

To propose a baseline for the artefact evaluation, we base our experiments on a machine running Ubuntu Desktop 20.04. On the local environment, the following packages are required:

- make, python3, git
- *any code editor*

Notable is that some of them might already be included with standard Linux-based OSs.

[1] The maintenance of a remote environment is expensive. We ask any interested reader in either following the video tutorials or to follow the official instructions on the official github repository.

On the remote environment, the following software is required:

- libc6-i386 python2.7-dev libtinfo5 libusb-dev libgconf-2-4 python3-pip
- *any code editor*
- Code Composer Studio IDE (CCS)

The listed elements are however already installed on the remote machine. Still, we provide the setup instructions for the reviewers to set up their own remote environment in case they had an available MCU.

A.4 Installation

Local Environment We leave the installation of a valid Ubuntu 20.04 desktop distribution to the reviewers. This is trivial due to the multitude of available tutorials online. Given that our fresh and minimal installation of Ubuntu comes with some packages already installed, we only perform the steps in Listing 1. These steps install the required packages, fetch the github repository and create an alias for a tool required in the later evaluation.

```
1 $ sudo apt install git make
2 $ cd ~/Documents/ && git clone https://github.com/
      MicheleGrisafi/PISTIS_AE.git
3 $ echo 'alias mspdump="~/Documents/PISTIS_AE/
      toolchain/compiler/msp430-gcc-9.2.0.50_linux64
      /bin/msp430-elf-objdump"' >> ~/.bashrc
4 $ source ~/.bashrc
```

Listing 1: Steps to install the required packages on the local environment

As already mentioned, we provide a "plug-n-play" VM to be used as remote environment by any reviewers. Still, for the sake of transparency and in the eventuality that the reviewer wanted to set up their own remote environment, in Listing 2 we provide the steps used to set it up.

```
1 $ sudo apt install libc6-i386 python2.7-dev
      libtinfo5 libusb-dev libgconf-2-4 python3-pip
2 $ pip3 install pyserial
3 $ cd ~/Downloads && wget https://dr-download.ti.
      com/software-development/
      ide-configuration-compiler-or-debugger/
      MD-J1VdearkvK/11.2.0.00007/CCS11.2.0.00007
      _linux-x64.tar.gz
4 $ tar -xvf CCS11.2.0.00007_linux-x64.tar.gz
5 $ ./CCS11.2.0.00007_linux-x64/ccs_setup_11
      .2.0.00007.run
6 $ echo 'PATH="/home/pistis/ti/ccs1120/ccs/eclipse:
      $PATH"' >> ~/.bashrc
7 $ source ~/.bashrc
```

Listing 2: Steps to set up the remote environment.

These steps download the required packages, fetch the CCS binary from the official TI website, extract it and install it. Finally, they add the installation directory to the Linux PATH. During the CCS installation, we should select the minimal installation and only select the "MSP430 ultra-low power MCUs" option. Afterward, we have to install the MSP430 toolchain from the official repository or link CCS with our version. We will proceed with the first option. In CCS, we should go to `Help/CCS App Center`, select the checkbox under MSP430 GCC and click `Install Software`. After the installation we can restart CCS.

A.5 Experiment workflow

This artefact evaluation contains two types of experiments. The first mainly allow the reviewers to evaluate the custom toolchain, the second allow them to inspect the run-time behaviour of Pistis.

Please be aware that for the second set of experiments, an SSH connection to our remote environment is required. Furthermore, we remind the reviewers that it is a single shared instance connected to a single MCU. As a consequence, they cannot operate simultaneously on the remote environment. We leave the organisational task to the reviewers.

Disclaimer: The debugging experience with Code Composer Studio, the official debugger for the MSP430 MCU, can be non-ideal and lead to inconsistent results. There might be executions with weird behaviours. This can be attributed to the state of the MCU and some internal CCS/debugger issues. The reader is kindly asked to follow the instructions as close as possible, sending us (michele.grisafi@unitn.it) an email in case of any issues. Finally, the interaction with the remote environment might not be ideal (slow and not quite responsive). We leave some video tutorials performed on the exact same setup on how to perform the experiments. If the reviewers cannot manage to operate on the remote environment, they are more than encouraged to check our result in such videos.

A.6 Evaluation and expected results

Code inspection The `README.md` file in the github repository contains the repository folder structure, with a description of the various content. The repository only contains the source files, which can be inspected at the reviewers discretion. The reviewer might inspect the various module composing the TCM, inside the `TCM/` folder, and the toolchain scripts, inside the `toolchain/` folder. The inspection of the source files can be skipped in favour of the following steps and evaluations. Still, given that Pistis is a complex software, we encourage the reviewer to inspect as much of it as possible. We authors remain available for any question on this matter.

Compiling the Trusted Computing Base The Trusted Computing Base (TCM) is the core of Pistis, containing both its basic functionalities and some Trusted Applications, i.e., additional features. To compile Pistis into a deployable we can follow the commands in Listing 3. Step (4) allow the inspection of the disassembly of the binary, while step (5) allow the inspection of code sections of the ELF file. We leave this data, which describes the TCM, for the more experienced and curios reviewers. Any interaction with the deployable file is more than welcome.

```
$ cd ~/Documents/PISTIS_AE/TCM
$ make clean && make
$ mspdump -D deployable.out > /tmp/dump.txt
$ cat /tmp/dump.txt
$ readelf -S deployable.out
```

Listing 3: Steps for the TCM compilation

A functioning toolchain One of the claim in the paper is a modified toolchain that transparently instruments the untrusted application code. This allow the new application image to be executed on a Pistis-enabled device. We ask the reviewers to perform a few steps to evaluate our toolchain (note that the reviewer is free to use the following indications as mere guidelines and perform his/her own tests). In particular, we will compile and inspect the instrumentation for a single application: XorCypher. Next, we describe the required steps for this evaluation. Each step is linked to the commands in Listing 4 (the number of the line will be included in parenthesis, e.g., (1)).

- Move to the `UpdateApplication` folder inside the repository (1) and clean it (2) to make sure no other residual file is present (traces of old compilations).
- Copy the source files of the XorCypher application in the `src` sub directory (3).
- Compile the application without using the modified toolchain (4). If the compilation was successful, the following message should pop out in the console: "*Metadata added –> created file deployable.out*". This informs us that our custom binary was created with the addition of some metadata (only required for the transmission of the binary).
- We can use mspdump[2] to retrieve the content of the binary (5). Mspdump is a disassembler for MSP430 binaries. Since mspdump can only read valid ELF files, and that our binary is a custom optimised format, we use mspdump on the original binary: `appWithNoMetadata.out`.
- Inspect the dump of the binary (6), where illegal instructions can be found. For instance, the reviewers can check for the presence of the `reta` instruction, which is not compatible with Pistis (i.e., it is an unsafe instruction). Alternatively, the assembly file in `UpdateApplication/asm/cryptoXor.s`[3] can be inspected (9).
- (Optional) Compare the size of the two binaries (7): the `deployable.out` and the `appWithNoMetadata.out`. It can be seen how our binary is considerably smaller, as the paper claims.
- Re-compile the application using our modified toolchain (8).
- Inspect the file as before (5)(6) and observe how there is no trace of `reta` instructions anymore: they have been virtualised.
- To have a better look at the instrumentation, open the assembly file `UpdateApplication/asm` which contains the new instrumented assembly code (9). The instrumentation will be contained within comments, e.g. starting from `";Old instruction: RET"` to the `";End safe sequence"`. Furthermore, observe the CFI NOP Slides inserted after each `CALL` statement. The reviewer is welcome to deeply inspect such assembly files.

```
$ cd ~/Documents/PISTIS_AE/UpdateApplication
$ make clean && rm src/* -rf && mkdir src
$ cp ../TestApps/XorCypher/xorCypher.c src/
$ make USE_NEW_LIB=0 VERIFY=0
$ mspdump -D appWithNoMetadata.out > /tmp/
    dump.txt
$ cat /tmp/dump.txt
$ stat -c%s deployable.out appWithNoMetadata.
    out
$ make clean && make libraries && make
$ cat asm/xorCypher.s
```

Listing 4: Steps for the toolchain evaluation

For the second part of this evaluation, we will show how Pistis toolchain rejects applications having illegal instructions, i.e., instructions trying to explicitly violate the access control policy enforced by Pistis. To demonstrate this, we crafted one such application containing a single illegal instruction: `BR #0x3400`. Such instruction is indeed trying to jump to the `0x3400` address which is in RAM, thus illegal[4]. Listing 5 show the required steps.

```
$ cd ~/Documents/PISTIS_AE/UpdateApplication
$ make clean && rm src/* -rf && mkdir src
$ cp ../TestApps/Malicious/rejectmalicious.c
    src/
$ make
```

Listing 5: Steps for the toolchain evaluation

If everything functioned properly, the compilation at the last step should output an error *"The compiled application*

[2]The alias was created during the installation phase

[3]Note that this assembly file does not contain the stdlib code.

[4]We recall that Pistis enforces a non-executable RAM.

has some unsafe code segments. Stop". This is because the toolchain found an illegal instruction.[5]

The reviewers are more than welcome to perform any variation of this test. For instance, they could try and compile the other applications or craft an application of their own. To do so, the only step that needs to be adapted is (3), where the reviewers should copy the source files of their liking.

Runtime verification in action One of the key features of Pistis is the ability to inspect any deployed binary at runtime, performing a verification. This step ensures that the binary has indeed been compiled with our custom toolchain, ultimately ensuring the presence of the instrumentation. To evaluate the runtime verification of the untrusted code by Pistis, we will use the debugging features of CCS. Given the necessity of a MCU, we provide the reviewers with an VM instance connected to a MSP430F5529LP board. Since CCS is a GUI-based application, we provide a video-tutorial on the various steps required for this evaluation: `https://youtu.be/tpEBLgRCVAU` . This should help the reviewer in performing the same evaluation. Nevertheless, we provide some guidelines on what we will need to do.

In this evaluation we will use the malicious application shown in listing 6. The application contains two lines of assembly: a `MOV` operation loading an address (pointing to RAM) into a CPU register, and a `BR` instruction jumping to that address (via the register). This application is malicious since it tries to jump to an address in RAM, thus violating the memory protection imposed by Pistis.

```
1  __asm("MOV #0x3400, R9");
2  __asm("BR R9");
```

Listing 6: Malicious application that tries to jump in RAM with a dynamic BR instruction.

For the evaluation we perform the following steps (also shown in the video tutorial):

- Compile the application using the `make VERIFY=0` command, which invokes a non-modified version of the compiler. This will produce an application binary without any instrumentation.
- Start a debugging session of Pistis using CCS. In this session we set a few breakpoints in some sensitive points in the code. Specifically, we want to break the execution when we reach either one of the following: verification passed, verification failed.
- Start the execution of Pistis, which will proceed with its RemoteUpdate feature and wait for an incoming image on the serial communication.
- Deploy the new application binary (without instrumentation) through our python deploy script.
- Observe how the second breakpoint is triggered: the verification fails and the application is not lunched.

This tutorial hence shows how Pistis bounds applications to our instrumentation, i.e., to using our toolchain. Pistis will only accept binaries which have indeed been compiled with our toolchain.

In the next tutorial, we will show how Pistis run-time memory protection protects the MCU from the malicious activity of an application compiled with our toolchain.

Memory protection in action To evaluate the memory protection offered by Pistis on the MCU we will use the debugging features of Code Composer Studio (CCS). These will enable a run-time debugging of the MCU, allowing us to check the operations of Pistis at run-time. For this purpose, we deploy the same application of listing 6. However, contrarily to previous experiment, we will instrument the malicious application with our custom toolchain[6]. This will allow it to be deployed, pass the verification and then be executed. However, since the unsafe instruction (the jump to a register) is indeed an illegal operation, this will be caught at run-time and the application will be stopped. Given that an interaction with a GUI is necessary for this step, we provide a video tutorial: `https://youtu.be/OhhJiyQC0bk`. Nevertheless, we report the main steps that we are going to do:

- Compile the application using the `make` command, which invokes a modified version of the compiler. This will produce an application binary with Pistis instrumentation.
- Start a debugging session of Pistis using CCS. In this session we set a few breakpoints in some sensitive points in the code. Specifically, we want to break the execution when we reach either one of the following: verification passed, verification failed, virtual safe BR function invoked. The latter is the function that checks all unsafe BR instructions in the code (which have been replaced by a call to this virtual safe function by our toolchain).
- Start the execution of Pistis, which will proceed with its RemoteUpdate feature and wait for an incoming image on the serial communication.
- Deploy the new application binary using our custom python deployer.
- Observe how the first breakpoint is triggered: the verification succeed and the application is lunched.
- Observe how the third breakpoint is reached: the application BR instruction is correctly virtualised.

[5] Note that illegal instructions are rejected right away, while unsafe instructions are virtualised. The latter cannot be rejected right away because their outcome depends on the run-time state of the MCU.

[6] Notably, the application is not rejected by our toolchain, but its unsafe instructions are instead virtualised.

- Observe how the safe BR function performs some security checks on the original instruction and stops the execution of the application, given that the original jump is illegal.

A.7 Experiment customization

The reviewers are encouraged to perform any experiment of their liking on the local environment. For instance, they could choose to execute or compile different applications, or even craft their own. However, given the scarce resources of the remote environment, we kindly ask them no to deviate from the provided instructions. Any modification could impede the work of the other reviewer. Moreover, the remote environment is provided with root access, thus allowing the reviewers to completely compromise it if they operate outside of our guidelines.

A.8 Notes

This artefact evaluation covers a few of the main functionalities of Pistis, showing its potential. Pistis is almost fully implemented (a few bugs and minor tweaks still to be addressed) and it has been fully tested and evaluated (as documented in the paper). However, the full evaluation is a cumbersome and time-demanding process requiring several technical ability. Furthermore, setting up a tutorial on how to properly test all of its features is even a more challenging task (especially considering the remote nature of the majority of the tests). For these reasons, this artefact only presents some of the possible tests that could be performed on the executable. The creation of complete tutorials is a future work.

Nevertheless, the repository contains a README.md file that describes in details some additional steps required to use Pistis. Note that this artefact document summarises only some of these steps, providing some techniques to evaluate it without owning the proper hardware.

A.9 Version

Based on the LaTeX template for artefact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

Our artifact is a docker image that provides the fully pre-installed SAPIC$^+$ platform. The SAPIC$^+$ platform for automated protocol security analysis allows to use multiple backends (TAMARIN, PROVERIF and DEEPSEC) from a single model.

We carried out a set of case-studies, described in Figure 7 of the paper, that are included in the docker and can be verified using the pre-installed platform.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Our artifact is a Docker Image.
- **Metrics:** Execution time, verification results (security proofs or attacks).
- **Output:** A csv file summarizing results.
- **Experiments:** Verification scripts.
- **How much disk space required (approximately)?:** 250MB for the docker image.
- **How much time is needed to prepare workflow (approximately)?:** A few minutes.
- **How much time is needed to complete experiments (approximately)?:** 2—3 hours.
- **Publicly available (explicitly provide evolving version reference)?:** Docker link.
- **Code licenses (if publicly available)?:** GNU GPL v3.
- **Archived (explicitly provide DOI or stable reference)?:** link to docker image or, alternatively, link to github repository.

A.3 Description

A.3.1 How to access

If docker is installed the artifact can be obtained by the following command:

```
docker pull robertkuennemann/sapicplusplatform
```

As SAPIC$^+$ is an extension of the TAMARIN prover is has been merged in the official develop branch of the repo and can be directly obtained from https://github.com/tamarin-prover/tamarin-prover. SAPIC$^+$ can then be installed by first installing Tamarin, and then Proverif v2.04 and DeepSec v2.0.0.

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

Docker.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Installation instruction for Docker are provided at https://docs.docker.com/engine/install/.

The image is obtained with

```
docker pull robertkuennemann/sapicplusplatform
```

and can then be browsed by running

```
docker run -it robertkuennemann/sapicplusplatform bash
```

A.5 Experiment workflow

Once inside the Docker, our case-studies can be reproduced by running two scripts in the `example` directory

- `./run-proverif-CS.sh`
- `./run-tamarin-CS.sh`

A.6 Evaluation and expected results

The scripts above execute all the case studies discussed in the paper (Figure 7), and store the results of either using Tamarin or Proverif to verify a given protocol. After completion, they should have created respectively a "examples/res-pro.csv" (PROVERIF results) and "examples/res-tam.csv" (TAMARIN results) files. Each line corresponds to one verification, using the format "protocol name; verification result; run time".

Note that the case studies need approximately 2–3 hours and 12GB to run. On OS X, Docker runs on a virtual machine with a builtin memory limit of 2GB, which must thus be increased to at least 12GB in the configuration pane located at 'Preferences/Resources/Advanced settings'.

Outside of a Docker, the PROVERIF script should complete in a few minutes on a standard laptop, while the TAMARIN script may take longer, but no more than one hour on a laptop. This may vary inside the docker depending on allocated resources.

We provide an additional docker image that is built from the previous one and by running the two scripts. It can be used to see the expected results by browsing the csv files:

```
docker pull robertkuennemann/sapicplusplatformbench
```

A.7 Experiment customization

Users familiar with protocol verification can use the docker image to verify new protocols. The image can be used to run SAPIC$^+$ with TAMARIN, PROVERIF or DEEPSEC on new examples. See the "README-platform" file in the docker image for more information.

A.8 Notes

N/A

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

The artifact discovers the vulnerability gap between manual models and automl models against various kinds of attacks (adversarial, poison, backdoor, extraction and membership) in image classification domain. It implements all datasets, models, and attacks used in our paper.

We expect the artifact could support the paper's claim that automl models are more vulnerable than manual models against various kinds of attacks, which could be explained by their small gradient variance.

A.2 Artifact check-list (meta-information)

- **Binary:** on pypi with any platform.
- **Model:** Our pretrained models are available on Zenodo (link). Follow the model path style `{model_dir}/image/{dataset}/{model}.pth` to place them in correct location.
- **Data set:** CIFAR10, CIFAR100 and ImageNet32. Use `--download` flag to download them automatically at first running. ImageNet32 requires manual set-up at their website due to legality.
- **Run-time environment:**

  At any platform (Windows and Ubuntu tested).

  'Pytorch' and 'torchvision' required. (CUDA 11.3 recommended)

  'adversarial-robustness-toolbox' required for extraction attack and membership attack.
- **Hardware:** GPU with CUDA support is recommended.
- **Execution:** Model training and backdoor attack would be time-consuming. It would cost more than half day on a Nvidia Quodro RTX6000.
- **Metrics:** Model accuracy, attack success rate, clean accuracy drop and cross entropy.
- **Output:** console output and saved model files (.pth).
- **Experiments:** OS scripts. Recommend to run scripts 3-5 times to reduce the randomness of experiments.
- **How much disk space required (approximately)?:** less than 5GB.
- **How much time is needed to prepare workflow (approximately)?:** within 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 3-4 days.
- **Publicly available?:** on GitHub.
- **Code licenses (if publicly available)?:** GPL-3.
- **Archived (provide DOI)?:** GitHub commit ade119d3c9aa1e851eba7db35f2de3c99eb0bf33.

A.3 Description

A.3.1 How to access

- **GitHub:** pip install -e .
- **PYPI:** pip install autovul
- **Docker Hub:** docker pull local0state/autovul
- **GitHub Packages:** docker pull ghcr.io/ain-soph/autovul

A.3.2 Hardware dependencies

Recommend to use GPU with CUDA 11.3 and CUDNN 8.0. Less than 5GB disk space is needed.

A.3.3 Software dependencies

You need to install `python==3.9, pytorch==1.10.x, torchvision==0.11.x` manually.

ART (IBM) is required for extraction attack and membership attack. `pip install adversarial-robustness-toolbox`

A.3.4 Data sets

We use CIFAR10, CIFAR100 and ImageNet32 datasets. Use `--download` flag to download them automatically at first running. ImageNet32 requires manual set-up at their website due to legality.

A.3.5 Models

Our pretrained models are available on Zenodo (link). Follow the model path style `{model_dir}/image/{dataset}/{model}.pth` to place them in correct location.

A.4 Installation

- **GitHub:** pip install -e .
- **PYPI:** pip install autovul
- **Docker Hub:** docker pull local0state/autovul
- **GitHub Packages:** docker pull ghcr.io/ain-soph/autovul

(optional) Config Path

You can set the config files to customize data storage location and many other default settings. View `/configs_example` as an example config setting.

We support 3 configs (priority ascend):

- **package (DO NOT MODIFY)**
  - `autovul/base/configs/*.yml`
  - `autovul/vision/configs/*.yml`
- **user**
  - `~/.autovul/configs/base/*.yml`
  - `~/.autovul/configs/vision/*.yml`
- **workspace**
  - `./configs/base/*.yml`
  - `./configs/vision/*.yml`

A.5 Experiment workflow

Bash Files

Check the bash files under `/bash` to reproduce our paper results.

Train Models

You need to first run `/bash/train.sh` to get pretrained models.

If you run it for the first time, please run with `--download` flag to download the dataset:

```
bash ./bash/train.sh "--download"
```

It takes a relatively long time to train all models, here we provide our pretrained models on Zenodo (link). Follow the model path style `{model_dir}/image/{dataset}/{model}.pth` to place them in correct location. Note that it includes the pretrained models for mitigation architectures as well.

Run Attacks

`/bash/adv_attack.sh`
`/bash/poison.sh`
`/bash/backdoor.sh`
`/bash/extraction.sh`
`/bash/membership.sh`

Run Other Exps

Gradient Variance
`/bash/grad_var.sh`
Mitigation Architecture
`/bash/mitigation_train.sh` (optional)
`/bash/mitigation_backdoor.sh`
`/bash/mitigation_extraction.sh`

Optionally, You can generate these architectures based on DARTS_V2 using `python ./projects/generate_mitigation.py`. We have already put the generated archs in `autovul.vision.utils.model_archs.darts.genotypes`. Note that we have provided the pretrained models for mitigation architectures on Google Drive as well.

For mitigation experiments, the architecture names in our paper map to:

- **darts-i:** `diy_deep`
- **darts-ii:** `diy_noskip`
- **darts-iii:** `diy_deep_noskip`

These are the 3 options for `--model_arch {arch}` (with `--model darts`)

To increase cell depth, we may re-wire existing models generated by NAS or modify the performance measure of candidate models. For the former case, we have provided the script to rewire a given model (link). Note that it is necessary to ensure the re-wiring doesn't cause a significant performance drop. For the latter case, we may increase the number of training steps in the single-step gradient descent used in DARTS.

To suppress skip connects, we replace the skip connects in a given model with other operations (e.g., convolution) or modify the likelihood of them being selected in the search process. Fro the former case, we have provided the script to substitute skip connects with convolution operations (link). Note that it is necessary to ensure the substitution doesn't cause a significant performance drop. For the latter case, we may multiply the weight of skip connect α_{skip} by a coefficient $\gamma \in (0,1)$.

Loss Contours

Take the parameter-space contour as an example. We pick the parameters of the first convolutional layer and randomly generate two orthogonal directions d_1 and d_2 in the parameter space. For simplicity, we set all each dimension of d_1 and d_2 to be either $+1$ or -1 in a random order and ensure that their orthogonality as $d_1 \cdot d_2 = 0$. We then follow Equation (12) in the paper to explore the mesh grid of $[-0.5, 0.5] \times [-0.5, 0.5]$ and plot the loss contour. A similar procedure is applied to plot the loss contour in the input space, but with the grid set as $[-0.2, 0.2] \times [-0.2, 0.2]$

A.6 Evaluation and expected results

Our paper claims that automl models are more vulnerable than manual models against various kinds of attacks, which could be explained by low gradient variance.

Training

(Table 1) Most models around 96%-97% accuracy on CIFAR10.

Attack

For automl models on CIFAR10,

- **adversarial:** (Figure 2) higher success rate around 10% (±4%).
- **poison:** (Figure 6) lower accuracy drop around 5% (±2%).
- **backdoor:** (Figure 7) higher success rate around 2% (±1%) and lower accuracy drop around 1% (±1%).
- **extraction:** (Figure 9) lower inference cross entropy around 0.3 (±0.1).
- **membership:** (Figure 10) higher auc around 0.04 (±0.01).

Others

- **gradient variance:** (Figure 12) automl with lower gradient variance around 2.2 (±0.5).
- **mitigation architecture:** (Table 4, Figure 16, 17) deep architectures (`darts-i, darts-iii`) have larger cross entropy for extraction attack around 0.5, and higher accuracy drop for poisoning attack around 7% (±3%) with setting of 40% poisoning fraction.

A.7 Experiment customization

Use `-h` or `--help` flag for example python files to check available arguments.



A Artifact Appendix

A.1 Abstract

Our paper contains a literature evaluation and a survey study with developers. One of the purposes of our literature evaluation was to extract relevant survey questions from the evaluated papers to design a questionnaire for our survey study. Therefore, in order to support our paper and make it more useful for the readers, we provide all the necessary artifacts available in a replication package. The complete replication package contains the screening and final survey questions we used, texts used in the recruitment emails or in the job posts, the consent forms, the formatted collection of questions we found in our literature evaluation, and additional result figures and tables.

A.2 Artifact check-list (meta-information)

- **How much disk space required (approximately)?:** Size on disk is 564 KB
- **Archived (explicitly provide DOI or stable reference)?:** https://doi.org/10.25835/wg7xhqmh

A.3 Description

In this section, we provide the descriptions of all the applicable subsections for our use case (i.e., "artifacts available" badge).

A.3.1 How to access

Our artifact can be accessed using the following URL: https://doi.org/10.25835/wg7xhqmh.

The complete replication package can be downloaded as a .zip file through the provided link. This replication package is hosted on the Research Data Repository of our university (data.uni-hannover.de).

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

N/A

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Our artifacts can be downloaded as a .zip file from the URL we provided in the section A.3.1. The file contains the following seven .pdf files:

1. **surveys.pdf:** This .pdf contains the screening and final survey questions we used
2. **recruitment-emails.pdf:** This .pdf contains the texts used in the recruitment emails
3. **job-posts.pdf:** This .pdf contains the texts used in the job posts
4. **consent-forms.pdf:** This .pdf contains the consent forms
5. **question-bank.pdf:** This .pdf contains the formatted collection of questions we found in our literature evaluation
6. **additional-figures.pdf:** This .pdf contains the additional result figures
7. **additional-tables.pdf:** This .pdf contains the additional result tables

A.5 Evaluation and expected results

We make the necessary artifacts available to support the literature evaluation and the survey study in our paper. Following is a checklist which represents how the provided artifacts support the paper:

- **Paper Section 3.2:** Section 3.2 in our paper details the literature survey. One of the contributions from this section is the formatted collection of survey questions we collected from the past papers. We provide this question collection document (*question-bank.pdf*) as one of our artifacts.
- **Paper Section 4:** Section 4 in the paper provides details of our comparative survey study. To support this section, we provide the screening and main surveys, texts used for participant recruitment and consent forms as artifacts (*surveys.pdf*, *recruitment-emails.pdf*, *job-posts.pdf* and *consent-forms.pdf*).
- **Paper Section 5:** The additional result figures and tables mentioned in Section 5 of the paper are provided as artifacts (*additional-figures.pdf* and *additional-tables.pdf*).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact is an implementation of the proposed mutual attestation mechanism, *i.e.*, MAGE, for Intel SGX. It includes an SDK library `libsgx_mage` that facilitates developers to use mutual attestation.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Developed and tested under Ubuntu 18.04.6 LTS.
- **Hardware:** An Intel CPU that supports Software Guard eXtensions (SGX), *e.g.*, Intel Core i5-6200U
- **How much disk space required (approximately)?:** 10 GB
- **How much time is needed to prepare workflow (approximately)?:** Couple of hours.
- **How much time is needed to complete experiments (approximately)?:** Couple of hours.
- **Publicly available?:** Yes, open-sourced on Github.
- **Code licenses (if publicly available)?:** BSD License

A.3 Description

A.3.1 How to access

`libsgx_mage`: https://github.com/donnod/linux-sgx-mage/tree/713fbd7479a37d1b768c615b3fd656c1774d9601

A.3.2 Hardware dependencies

An Intel CPU that supports Software Guard eXtensions (SGX) is needed for the evaluation. Disk space requirement: 10 GB.

A.3.3 Software dependencies

This artifact is developed and tested under Ubuntu 18.04.6 LTS.

A.4 Installation

Installation instructions can be found in the README.md file.

A.5 Experiment workflow

- Install `libsgx_mage` following the instructions in the README.md file.
- Go to sub-folder: SampleCode/MutualAttestation, and run make in the terminal to build the application.
- Run the built binary.

A.6 Evaluation and expected results

Main claim. A group of enclaves could derive the other's measurements without trusted third parties.

Steps. The example code builds three enclaves, each of which could derive the others' measurements. Particularly, in the output of the executable, each enclave prints its own measurements and three derived measurements.

Key and expected results. Three enclaves output the same list of three derived measurements. Each of the derived measurement is the same as one of the enclaves' own measurements.

The performance evaluation instructions and expected results are in the "Artifact Evaluation" section of the README.md file.



1 Artifact Appendix

1.1 Abstract

The evaluated artifact includes the prototype implementation of ELASTICLAVE that we have presented in the paper. We have publicly released it on GitHub.

Running the full set of experiments take significant long time and relies on the AWS EC2 platform. Therefore, we also provide the option to run them with QEMU, which, though inaccurate for performance evaluation, serves well as a quick way to test the system functionally. This option only requires an x86-64 Linux system with Docker installed.

1.2 Artifact check-list (meta-information)

- **Program:** IOZone (included)
- **Compilation:** GCC cross compiler targeting RISC-V 64 (included)
- **Run-time environment:** Linux (Ubuntu 20.04 LTS recommended) with Docker
- **Hardware:** x86-64
- **Metrics:** Execution time
- **Output:** Performance numbers (execution time) in console log files. Results are the differences in the performance among different solutions
- **Experiments:** Run the benchmarks with the scripts we have prepared. Compare the performance numbers produced with different solutions and the difference should be on the same order of magnitude as the results reported in the paper
- **How much disk space required (approximately)?:** 20 GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 30 hours
- **Publicly available?:** Yes

1.3 Description

1.3.1 How to access

This artifact is publicly released on GitHub[1]. The commit hash of the evaluated version is 29aab39.

1.3.2 Hardware dependencies

It is necessary to run the artifact on an AWS EC2 and use FireSim to obtain accurate performance data. For evaluation of the functionality, any modern x86-64 Linux platform should suffice. The required disk space is approximately 20 GB.

[1]The main repository (which references more repositories as submodules): https://github.com/jasonyu1996/elasticlave

1.3.3 Software dependencies

This artifact is expected to run on any GNU/Linux distribution with Docker installed.

1.4 Installation

We provide two options to run the artifact.

The first option requires running FireSim on an AWS EC2 F1 instance, and hence can incur significant monetary cost. In addition, it takes much longer to run the experiments than the second option. Since this is the option that provides cycle-accurate simulation, it is necessary if the goal is to evaluate the performance of the system and reproduce the experimental results.

The second option is to emulate the system on QEMU. It does not incur extra cost and consumes less execution time. This option is unable to produce accurate performance data and is only suitable for testing the functionality.

We have automated most part of the installation process. Below are the installation instructions for both options.

FireSim. On your AWS EC2 instance with FireSim set up, clone the repository and checkout to the evaluated snapshot:

```
git clone https://github.com/jasonyu1996/elasticlave.git
cd elasticlave
git checkout 29aab39
```

Pull the submodules recursively:

```
git submodule update --init --recursive
```

Build:

```
./docker.sh
./docker-run.sh ./make-firesim.sh
./docker-run.sh ./make-firesim.sh image
```

Launch the simulated system:

```
./run-firesim.sh
```

It might help to understand what happens under the hood in the script executed above.

The script first launches FireSim:

```
firesim launchrunfarm && firesim infrasetup && \
    firesim runworkload
```

After this is finished, it logs into the newly launched F1 instance:

```
ssh RUNFARM_IP
```

, where `RUNFARM_IP` is the IP address of the F1 instance as reported in `firesim runworkload`.

The script then connects to the terminal of the simulated system:

```
screen -r fsim0
```

When the simulation ends, the script terminates the F1 instance: Terminate the F1 instance:

```
firesim terminaterunfarm
```

Below are instructions for use inside the connected terminal of the simulated system.

The login is `root` and the password is `sifive`.

To run the benchmarks, execute the following inside the shell of the prototype system:

```
insmod keystone-driver.ko
./tests.ke
```

There are also individual benchmarks that are not included in `tests.ke`. To run them, execute the scripts in the individual folders.

After the benchmark execution completes, you can end the simulation through:

```
poweroff -f
```

A log of the data on the terminal can be found inside `FIRESIM_FOLDER/deploy/results-workload`.

QEMU. Install Docker following the instructions on the official website.

Clone the repository and run the build scripts:

```
git clone https://github.com/jasonyu1996/elasticlave.git
cd elasticlave
git checkout 29aab39
git submodule update --init --recursive
./docker.sh
```

Run the artifact:

```
./docker-run.sh ./run.sh
```

The login is `root` and the password is `sifive`. The password is `sifive`.

To run the benchmarks, execute the following inside the shell of the prototype system:

```
insmod keystone-driver.ko
./tests.ke
```

There are also individual benchmarks that are not included in `tests.ke`. To run them, execute the shell scripts (`*.sh`) in the individual folders.

1.5 Experiment workflow

As described in Section 6, The experiments involve a range of benchmarks which are run on our prototype ELASTICLAVE implementation. The benchmarks involve data sharing across enclave boundaries, and the total running time with the ELASTICLAVE model is compared against that with the traditional spatial isolation model, as well as the running time when they are run in a native Linux environment without the protection of a TEE.

The prototype system runs an unmodified Linux kernel (with a driver for enclave management). Each benchmark includes both enclaves and untrusted code which runs as the host process and launches the enclaves. For using the spatial isolation model, the untrusted code is also responsible for marshalling messages.

1.6 Evaluation and expected results

The key claims made in this paper include:

1. Compared to the spatial ShMem model, our ELASTICLAVE implementation achieves 1–2 orders of magnitude better performance for data sharing. The overhead of ELASTICLAVE is about 10% compared with native execution without a TEE;
2. ELASTICLAVE incurs modest TCB and hardware complexity impact.

This artifact can be used to verify the following key results that support the above claims:

1. The performance comparison among ELASTICLAVE, the spatial ShMem model, and native execution for data sharing on synthetic benchmarks and IOZone (corresponding to Figures 6, 7, 8, 10, and 11). This supports Claim 1 above.
2. The TCB increase of ELASTICLAVE over Keystone (corresponding to Table 4). This supports Claim 2 above.

1.6.1 Performance

To obtain accurate performance numbers, it is necessary to run the benchmarks using FireSim. See Section A.4 for details. The results obtained from this artifact are expected to reflect the same patterns as in Figures 6, 7, 8, 10, and 11 as well as the associated descriptions in Section 6.1.

IOZone. Set `TESTS=iozone` in `tests/tests/mkconfig.mk`, rebuild the benchmarks with `./docker-run.sh ./make-firesim.sh image`. and execute `./tests.ke` in simulation. This runs IOZone with ELASTICLAVE. To run it with the baseline spatial ShMem model or a native Linux setting, set `NATIVE_TESTS` or `BASELINE_TESTS` to `iozone` instead.

Thread synchronization. Set `EXTRA_TESTS` to `lock` (spinlock with ELASTICLAVE), `lock-futex` (futex with ELASTICLAVE, `lock-spatial` (spinlock with the spatial ShMem model), `lock-native` (futex without TEE) and rebuild the benchmarks. Then execute `./tests.ke <thread-count> <work-amount>` in simulation. To get the numbers reported in the paper, supply 2 as `thread-count` and vary `work-amount` from 12800 to 3276800.

Data sharing patterns. The names of the corresponding synthetic benchmarks start with `icall-`, followed by the names of the patterns (`consumer`, `server`, and `proxy-3`). Name endings indicate whether the benchmarks are run with the spatial ShMem baseline (`spatial`), ELASTICLAVE without exclusivity support (`ne`), or full ELASTICLAVE (otherwise). To run the benchmarks, open the file `tests/tests/mkconfig.mk`, add the benchmark names in the line that starts with `EXTRA_PACKS`, and rebuild the benchmarks. The available benchmark names can be viewed in `tests/tests`.

1.6.2 TCB Increase.

To measure the TCB increase over Keystone, download the revision of the original security monitor and enclave runtime from Keystone[2]. Use `diff -x '.*' -Nwr <old-dir> <new-dir> | diffstat` to compare the directories `riscv-pk` and `sdk/rts/eyrie` with them and pipe the results to. The sums of insertion and modification numbers are below the numbers reported in Table 4.

1.7 Experiment customization

You can adjust the benchmarks to be included in each run of the artifact. To achieve this, edit the file `tests/tests/mkconfig.mk`, and add the names of the benchmarks you want to run.

[2] https://github.com/keystone-enclave/riscv-pk/tree/5b3d71 and https://github.com/keystone-enclave/keystone-runtime/tree/87351c



A Artifact Appendix

A.1 Abstract

Minefield is a probabilistic undervolting protection for SGX enclaves implemented via a compiler extension. The general idea is to place instructions highly susceptible to undervolting faults between regular instructions. In the artifact evaluation, we include all the tools needed to reproduce each result of the paper to follow the conclusion of our mitigation. First, we provide the instruction finding framework that automatically scans the x86 instruction set for instructions susceptible to undervolting faults. Second, we show a benchmark for the minimal time between voltage transitions. Third, we include the compiler infrastructure to automatically generate hardened enclaves and the required modifications to the SGX-SDK. Finally, we provide the tools to reproduce the performance, size, compile-time, and detection rate benchmarks of Minefield. Due to the nature of the paper, we require Intel hardware that supports SGX and a runtime environment where possible data corruption is *acceptable*. We recommend a clean installation of Ubuntu 20.04, with Intel CPUs between the 6^{th} and 10^{th} generation. Furthermore, if applicable, undervolting faults will lead to repeated system freezes during the profiling phase. Therefore, an automatic way to restart the system would be beneficial.

A.2 Artifact check-list (meta-information)

- **Program:** The used programs are provided, or how to install them is described.
- **Compilation:** We require a modified Clang 11 compiler. Download and build scripts are provided.
- **Transformations:** We provide the patches used to allow compilation of the SGX-SDK with Clang.
- **Data set:** We provide the framework to use the `https://uops.info` x86 instruction-set list.
- **Run-time environment:** Requires a native Linux installation that supports SGX, and we strongly recommend Ubuntu 20.04. The provided installation scripts require internet access.
- **Hardware:** Intel CPUs with SGX support between the 6^{th} and 10^{th} generation and MSR `0x150` available. Undervolting-based faults are highly dependent on the actual hardware and even differ between cores on the same CPU. We recommend one of the CPUs of the paper.
- **Execution:** For executing the benchmarks, we require a stable frequency, isolated cores, a modified grub command line, and software-based undervolting.
- **Security, privacy, and ethical concerns:** Due to the undervolting **data-corruption** can occur on the used system.
- **Metrics:** The benchmarks report performance in iterations per second, faulting points in mV, execution time in seconds, code size in bytes, and detection rate factors.
- **Output:** The resulting outputs are CSV files. We provide visualization scripts where possible.
- **Experiments:** We include installation scripts and readmes describing the process and how to execute the benchmarks.
- **How much disk space required (approximately)?:** 4-5 GB
- **How much time is needed to prepare workflow (approximately)?:** 3-4 hours
- **How much time is needed to complete experiments (approximately)?:** 1-5 days depending on the depth of the analysis.
- **Publicly available (explicitly provide evolving version reference)?:** `https://github.com/iaik/minefield`
- **Code licenses (if publicly available)?: MIT**
- **Archived (explicitly provide DOI or stable reference)?:** `https://github.com/iaik/minefield/tree/ae`

A.3 Description

A.3.1 How to access

Check out the Git repository from `https://github.com/iaik/minefield` and follow the provided readmes.

A.3.2 Hardware dependencies

We require Intel CPUs which support SGX and have an available software undervolting interface (MSR `0x150`) available. We recommend CPUs between the 6^{th} and 10^{th} generation and recommend a desktop CPU shown in the paper. Our experience showed that the susceptibility to undervolting faults is highly dependent on the used hardware and even differs across cores from the same CPU. We recommend a system with physical access as undervolting faults will repeatedly crash the system and lead to system freezes.

A.3.3 Software dependencies

We strongly recommend Ubuntu 20.04 as it has official support for SGX, and we tested all the provided tools there. The components of the paper have to be built from source, hence the systems requires tools for compiling software (`build-essentials` on Ubuntu). Access to MSRs via the `msr-tools` interface is also necessary. Finally, we require a setup that allows frequency pinning via `cpupower` to fix the frequency at a given operating point during the undervolt.

A.3.4 Data sets

To speed up the finding of the susceptible instructions, we provide our found faultable instruction data set in the repository. Furthermore, we rely on the complete x86 instruction set list from `https://uops.info`, which is automatically used in the framework.

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

During our experiments with undervolting, we observed **data corruption** in recently used files. Therefore, we highly recommend a fresh installation with an operating system image not used for personal or important data. We *never* observed persistent damage on the hardware used for undervolting. However, we cannot ensure that this is generally the case, but we find it highly unlikely to damage the used hardware.

A.4 Installation

Follow the readmes in the top-level directory, which will guide you through installing all the necessary tools and components of the paper. The installation scripts are written in bash and *should* automate most of the process. However, we cannot rule out that some parts might need manual adjusting, and therefore, knowledge of C, C++, python3, bash, and Makefiles is beneficial. Furthermore, due to the enormous complexity of SGX, some packages might need manual installation if not found correctly.

A.5 Experiment workflow

After building the components for the benchmarks, they can be executed via scripts for a given *placement density*. These scripts should be executed with a fixed frequency to allow a fair comparison between the runs. The benchmark results are exported in the CSV format, and we provide additional scripts to convert the measurements into relative overhead percentages with respect to the baseline.

A.6 Evaluation and expected results

The reproduced results from Table 1 and Table 2 should show that `imul` is, across multiple CPUs, the instruction most susceptible to faults. Some concrete instances might require extended instructions to detect the fault at the highest undervolting point correctly. With this assumption, the compiler extension can rely on `imul` as trap instruction.

For the performance results, we should see a nearly linear performance decrease (Figure 8) and a rising code size (Figure 10) when increasing the *placement density*. Some benchmarks are more affected by the *placement density* than others. For the mbedTLS (Figure 9) benchmark, some configurations with different key lengths and disabled redundancy checks in the library itself show better performance as the baseline depending on the number of leading zeros in the key. The compile-time (Figure 11) should also rise with increasing *placement density*. However, the absolute time increase should be minimal.

Finally, we provide test enclaves to test the detection rate of the mitigation (Figure 6) in the worst-case scenario and a more realistic scenario when protecting mbedTLS (Figure 7).

A.7 Experiment customization

Since the undervolting offset is highly dependent on the hardware and even the core executing the code, some benchmarks might need manual adjustment. The instruction finding framework automatically detects system freezes when using our remote system with a remote power switch. The overall runtime of the performance benchmark can be adapted via the number of runs.

A.8 Notes

Undervolting faults are highly dependent on the used systems. Even our two identical systems from Table 2 show different faulting behavior. Furthermore, we observed different undervolting offsets on cores of the same CPU. Therefore it is likely that the undervolting-related results from the artifacts differ.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



H Artifact Appendix

H.1 Abstract

In this artifact, we provide datasets and software tools related to our paper "Anycast Agility: Network Playbooks to Fight DDoS" [9]. Our artifact contains several datasets generated from our anycast experiments and analysis. Our datasets provide a snapshot of the results that we generated during our experiments. Some of our experimental results are dependent on the current state of the network interconnections and policies. However, due to the anycast stability, we expect to get similar results if we redo the experiments now. Our published datasets support our key results and are publicly available. We also provide tools and scripts that can be useful for other researchers.

H.2 Artifact check-list (meta-information)

- **Algorithm:** We provide an algorithm to select the best routing option from a BGP playbook containing multiple routing options and their impacts over traffic distribution (Section 3.4.2 of the paper [9]). We provide a working Python script for this selection algorithm. We include instructions about this tool in our anygility tool page [10].
- **Compilation:** We use shell/python script and java program for our tools. One needs to install Python and Java to run our tools. We depend on Verfploeter software, and we mention a series of other dependencies in the software READMEs [10].
- **Binary:** Some of our tools require extra binary files. We include those binary files with our software package, and provide instructions.
- **Data set:** We provide several datasets generated from our experiments and analysis [11]. Some software tools require extra datasets to run (e.g. IP hitlist). We include a sample dataset file with the software tool.

  However, we do not include large data files with our software tools. But these datasets can be downloaded separately (we provide the instruction in §H.3.1).
- **Run-time environment:** We tested our tools in Linux operating system. Peering toolbox on Fedora 34, and Tangled on Ubuntu 18.04 LTS and macOS 12. In some cases, our tools require root access. Our tools notify the users when it needs root access.
- **Run-time state:** Our key idea related to network playbook (Section 3.1 and 6.4 of the paper [9]) is dependent on the network interconnections and policies. We include the dates of experiments in our datasets. Since anycast is stable, we expect a similar outcome if we rerun the experiment.
- **Execution:** Some of our tools might need a long time to run. For example, our automated playbook builder announces different routing configurations, runs Verfploeter, and captures traces after a fixed interval. If we consider the whole process from measurement to playbook for 7 sites, it takes around 27-35 hours. For 3 sites it was around 17-24 hours. If we have more sites, or more routing policies, it would take even more time.
- **Security, privacy, and ethical concerns:** In the required cases, we anonymize IP addresses to prevent IP disclosure. As an example, we anonymize IP addresses in the DDoS attack datasets. For privacy reasons, we restrain ourselves from sharing certain attack data from Dutch national scrubbing center, and from an enterprise.
- **Metrics:** We provide datasets related to anycast catchments and DDoS attacks. Each dataset reports different metrics. We provide the details of these metrics in our README files. Our README files are included with the dataset packages.
- **Output:** We provide experimental outputs from Tangled and Peering testbeds. Tangled provides the measurement output in csv format while Peering provides raw captured traces in pcap format. These data files are parsed to generate output files in human-readable formats or graphs. The graphs are built using jupyter notebook and gnuplot scripts. We provide these scripts in our dataset webpage [11].
- **Experiments:** We provide scripts to automatically announce different routing configurations in both Peering and Tangled testbeds. We provide our generated datasets from these experiments. We provide some sample data to test our route selection process independent from running the whole measurement process.
- **How much disk space required (approximately)?:** Software tarballs are about 500KB. Our datasets related to the anycast experiments require around 100 GB disk space. Our attack datasets are large since we provide the whole day traffic captures (around 500 GB each). As our datasets are large, a user can download a portion of the datasets.
- **How much time is needed to complete experiments (approximately)?:** Some of the experiments may take a whole day (building a playbook with all routing options). Measurement process can take days depending the chosen measurement. Our decision maker can take decision within seconds. Parsing tools may need different times depending on the data size.
- **Publicly available (explicitly provide evolving version reference)?:** Our evolving datasets and software tools are publicly available at `https://ant.isi.edu/datasets/anycast/anycast_against_ddos/index.html`.
- **Code licenses (if publicly available)?:** Our tools are free; so anyone can redistribute it and/or modify it under the terms of the GNU General Public License, version 2, as published by the Free Software Foundation. We include this license notice with every tools that we make publicly available.
- **Data licenses (if publicly available)?:** We follow the data sharing policy through the participation of the LACREND project in the DHS IMPACT program [5].
- **Archived (explicitly provide DOI or stable reference)?:** Our stable reference for this artifact is here: `https://zenodo.org/record/6473023` with DOI 10.5281/zenodo.6473023.

H.3 Description

We provide datasets and tools for measuring anycast agility against DDoS. Our datasets are available upon request [5]. We provide datasets about the traffic distribution after BGP changes in testbeds, attack data from a DNS root server and from a national scrubbing

Software tools	Software dependencies	Software source	Dataset dependencies	Dataset source
Traffic Estimator	Java	openjdk-11.0.13	pcap traces	With dataset
	tshark	Wireshark	RIPE IPs	Included
playbook builder	Access to Peering	Required Testbed access	Hitlitst	With dataset
	Pinger	Provided + open source		
playbook tuner	Python	Python 3.10.2	Playbook	Included
			Load	Included
load_parser+ ParsingLoad	shell+Java	openjdk-11.0.13	Dataset dir. with pcaps	With dataset
	pingextract	Provided + open source	Load file	With dataset
BGPTuner	Python bgptuner-requirements.txt	Python 3.8	Playbook with specific site list	Included
measurement scripts + tangler-cli	Bash Python Verfploter ExaBGP Access to Tangled	Bash 4.4 Python 3.8 Verfploter 0.1.42 ExaBGP 4.1.2	—	—
vp-cli	Python	Python 3.8	Verfploter 0.1.42 files	Included
make-playbook	Python	Python 3.8	stats files	Included
run-playbook	Python ExaBGP Access to Tangled	Python 3.8 ExaBGP 4.1.2	Routing Playbook	Included

Table 1: Software tools dependencies.

center, other data related to anycast catchment stability, and other supporting data for our software tools. We provide codes for traffic estimation, for reproducing experiments, and for parsing the collected data.

H.3.1 How to access

Our datasets are available from the institutional storage system [6]. We provide the datasets based on requests [5]. After getting a request, we provide the download instructions. Our software tools will be available to download from its own webpage [10].

H.3.2 Hardware dependencies

Our whole uncompressed datasets size is over 1 TB. However, a user can download the partial datasets [6]. An interested user may want to look over the meta data of each dataset (using the README files), and keep the required amount of free storage.

H.3.3 Software dependencies

We provide several tools for different purposes [10]. We tested our software tools in Linux operating system. Some of our tools are dependent on external data sources and binaries. In most cases, we provide a sample data source with the package, and for other cases one can download the datasets with our released dataset. We provide the required binaries with our tools. One might need to install dependencies like Python or Java. We detail dependencies on (Table 1).

H.3.4 Data sets

We provide a full list of datasets in our web page [11].

We release datasets related to catchment distribution after routing configuration changes. We announce different BGP options, run Verfploeter to ping millions of responsive targets, and then capture the responses at every site. Our dataset includes raw pcap files captured from these measurements, and parsed data files in human-readable format.

We also provide DDoS attack data collected from B-root and Dutch national scrubbing center from 2015 to 2021.

Within other datasets, we provide datasets for anycast stability, and other supporting datasets to run our software tools.

The READMEs for these datasets are available with the dataset package.

H.3.5 Models

N/A

H.3.6 Security, privacy, and ethical concerns

We see no privacy concerns with our shared datasets. In cases like the DDoS attack data, we only share the /24 prefixes to hide the exact IP.

H.4 Installation

Instructions for running the tools are available in the webpages [10].

H.5 Evaluation and expected results

We provide the key results of the paper by mentioning the figures and tables, and list the corresponding datasets and tools in Table 2. Next, we list the key results, then we describe how can one get these results, and possible variations in the results. Please check the detailed steps to regenerate the graphs from the provided datasets.

H.5.1 Results with traffic estimation:

We propose a new technique to estimate the true offered load when we have loss in the upstreams (Section 3.3 and 4 [9]). We show our traffic estimation technique works well with the real world-attack events. For traffic estimation, we provide a tool named *TrafficEstimator* [14]. Using our traffic estimation tool, we show that we can correctly estimate the true offered load for real-world DDoS events.

To reproduce the same result, one needs to feed the attack traces to our program (provided as attack data in peering dataset [12]). One needs to have the pcap traces that we used, and needs to install tshark (with Wireshark) to feed the traffic content to our program. We used tracefiles for 2015-11-30 and 2016-06-25 events. A user needs to know the attack start time to use the right pcap files to observe the estimation outputs. The provided README tells the attack start time. We also need to provide a list of RIPE IPs that our program will use (already provided with the tool). We provide the instructions for running this tool in our webpage [14].

If running correctly, one can regenerate the same results that we showed in the paper. Depending on the start and end time of the attack trace, we might get a slightly different estimation. But on average we expect to get the same results.

Detailed steps:

We show the results generated for 2015 and 2016 events. This covers Figure 4, Figure 12, and Table 1. We use the following datasets:

1. Non-attack traffic 2015: B_Root_Anomaly-20151130/29/20151129-065024-00175689.pcap.xz,
2. Non-attack traffic 2016: B_Root_Anomaly-20160625/24/20160624-200008-00356777.pcap.xz,
3. Attack traffic 2015: B_Root_Anomaly-20151130/30/20151130-065209-00177422.pcap.xz,
4. Attack traffic 2016: B_Root_Anomaly-20160625/25/20160625-221823-00357641.pcap.xz,
5. RIPE IPs 2015: ripe-ips-2015-11-30.txt (provided with the tool),
6. RIPE IPs 2016: ripe-ips-2016-06-25.txt (provided with the tool).

The first step is to calculate the RIPE traffic rate during normal period (known-good traffic - normal column of Table 1). To get this value, we feed non-attack traffic to our estimator to get the RIPE traffic rate during normal period. We use the following command to get this:

For 2015 event: xzcat B_Root_Anomaly-20151130/29/20151129-065024-00175689.pcap.xz | sudo tshark -r - -T fields -e frame.time_epoch -e ip.src | java -jar TrafficEstimator.jar ripe-ips-2015-11-30.txt 192.228.79.131,2001:500:84::9077:f4f0

For 2016 event: xzcat B_Root_Anomaly-20160625/24/20160624-200008-00356777.pcap.xz | sudo tshark -r - -T fields -e frame.time_epoch -e ip.src | java -jar TrafficEstimator.jar ripe-ips-2016-06-25.txt 192.228.79.62,2001:500:84::ad9b:d590

Please wait for some time to see the generated output in the command prompt. The given addresses (192.228.79.* and 2001:500:84::*) are the B root server addresses (different because of the different anonymization keys). This command will generate an output like:

2015: “Time diff: 5.01 Counter-packets: 193 Rate: 38.47” 2016: “Time diff: 5.06 Counter-packets: 195 Rate: 38.52”

Key results [9, 11]	Shared datasets	Related tools
Figure 3	sample dataset provided with the tool	*tangled tools [15]* bgp-tuner
Figure 4, Table 1, Figure 12	*peering and root DNS dataset [12]* B_Root_Anomaly-20151130 B_Root_Anomaly-20160625	*TrafficEstimator and selection tools [14]* TrafficEstimator
Figure 5	*peering and root DNS dataset [12]* anycast_catchment_distribution-20200224: prepending (3 sites) 2020-02-24	*peering tools [15]* playbook_builder load_parser ParsingLoad
Figure 6	*tangled dataset [13]* Usenix_anygility_5_sites_2022-03-24_NEW	*tangled tools [15]* measurement scripts tangler-cli, vp-cli Anygility-Tangled-Catchment-load-distribution.ipynb
Figure 7	*peering and root DNS dataset [12]* anycast_catchment_distribution-20200224, community (3 sites) 2020-02-25	*peering tools [15]* playbook_builder load_parser ParsingLoad
Figure 8	*tangled dataset [13]* community dataset (3 sites)	*tangled tools [15]* measurement scripts tangler-cli, vp-cli
Table 5, Table 6	*peering and root DNS dataset [12]* anycast_catchment_distribution-20200224: prepending (3 sites) 2020-02-24, community strings (3 sites) 2020-02-25, poisoning (3 sites) 2021-04-09	*peering tools [15]* load_parser ParsingLoad
Figure 9	*peering and root DNS dataset [12]* anycast_catchment_distribution-20200224: prepending (3 sites) 2020-02-28	*peering tools [15]* playbook_builder load_parser ParsingLoad
Figure 10	*peering and root DNS dataset [12]* anycast_catchment_distribution-20200224: prepending (3,5,7 sites) 2020-02-24, 2020-04-07, 2020-04-08 Community (3, 5, 7 sites) 2020-02-25 and 2020-04-19	*peering tools [15]* playbook_builder load_parser ParsingLoad
Table 7	*peering and root DNS dataset [12]* anycast_catchment_distribution-20200224: baseline (3 sites) 2020-02, 2020-04, and 2020-06	*peering tools [15]* load_parser ParsingLoad
Figure 11	*peering and root DNS dataset [12]* B_Root_Anomaly_message_question-20170306	*peering tools [15]* ParsingLoad TimeBasedPrefixLoad AnycastSiteLoad
Figure 13	*peering and root DNS dataset [12]* anycast_catchment_distribution-20200224: poisoning (3 sites) 2021-04-09	*peering tools [15]* load_parser ParsingLoad
Figure 14	*tangled dataset [13]* poisoning dataset (3 sites)	*peering tools [16]* measurement scripts tangler-cli, vp-cli
Figure 15	*peering and root DNS dataset [12]* anycast_catchment_stability-20210701	-
Figure 16	*peering and root DNS dataset [12]* B_Root_Anomaly_message_question-20200214 B_Root_Anomaly_message_question-20210528	*peering tools [15]* ParsingLoad TimeBasedPrefixLoad AnycastSiteLoad

Table 2: Paper key results with datasets and tools. We provide the scripts to generate the graphs for our key results in our webpage [11].

We waited until 5 s to fix the final rate of the RIPE IPs. This rate is the cumulative rate measured from the start time. known-good traffic - normal column from Table 1 has a similar value.

The second step is to run the same TrafficEstimation java utility to find the estimated rate. We run the following commands for this:

2015 event: xzcat B_Root_Anomaly-20151130/30/20151130-065209-00177422.pcap.xz | sudo tshark -r - -T fields -e frame.time_epoch -e ip.src | java -jar TrafficEstimator.jar ripe-ips-2015-11-30.txt 192.228.79.131,2001:500:84::9077:f4f0 38.47

2016 event: xzcat B_Root_Anomaly-20160625/25/20160625-221823-00357641.pcap.xz | sudo tshark -r - -T fields -e frame.time_epoch -e ip.src | java -jar TrafficEstimator.jar ripe-ips-2016-06-25.txt 192.228.79.62,2001:500:84::ad9b:d590 38.52.

Please note that this command has an extra parameter (38.47 and 38.52) which we got from the previous command outputs. This command will generate two types of output lines. For 2015 event, we are showing a snapshot after 20 s, and for 2016 event we are showing a snapshot after 42 s.

2015 event output:

Time diff: 19.99 Counter-packets: 37 Rate: 1.85 1448866349.106 Count-packets: 1604914 Observed rate: 320982.8 Estimated: 6674713.88

2016 event outptut:

Time diff: 41.98 Counter-packets: 14 Rate: 0.33

1466893148.1316 Count-packets: 451957 Observed rate: 90370.72 Estimated: 11186300

Our program shows the RIPE rate when it finds new RIPE IPs in DNS traffic (starting after 1 minute). The observed rate line is printed at every 5 s. So, the users normally observe more number of first line.

The first line for 2015 event indicates that after 20 s during the attack period, our program receives 37 RIPE packets at a rate of 1.85 RIPE packets/s. This value corresponds to the known good traffic - observed column value from Table 1. Dividing by the prior normal rate of 38.47, we get the access fraction value, α. The first line for 2016 event indicates that the program gets 14 RIPE packets within 41.98 s with a rate of 0.33 RIPE packets/s. This value indicates the known good traffic - observed column value from Table 1. Dividing by the prior rate of 38.52 RIPE packets/s, we get the value of access fraction, α in Table 1. Please note that, because of a different RIPE IP list and measurement start time (using different pcap files), we are getting a slightly different value than what we have in the table.

Our program generates the second line at every 5 s. This line indicates the timestamp at every 5 s, packet count within that 5 s, the observed rate (packet count / 5.0), and the estimated traffic rate (observed rate / α). This observed rate corresponds to the offered load during attack - observed rate column of Table 1. For 2015 event, the sample output value is close to 0.32M packets/s, and for 2016 event this value is 0.09M packets/s. These two values are similar to what we have in Table 1 (0.37 and 0.10). A user will observe variable rates at different times. This observed rate is then divided by the calculated access fraction (α) to get the estimated offered load—offered load during attack - estimated column ($\sim$6.6M queries/second for 2015 event and $\sim$11M queries/second for 2016 event), which is close the reported rate of 5M queries/second and 10M queries/second, respectively [7, 8]. We use the estimated values from our TrafficEstimation program to generate the graphs—Figure 4 and Figure 12. Depending on the attack start time and RIPE IPs, the estimated values may vary slightly but we expect to get a similar trend. The offered load during attack - normal column indicates the normal traffic rate at a given time which we can measure from B root traffic (TrafficEstimator tool can measure this; we just need to feed the normal traffic with the known RIPE rate parameter) but we are skipping this detail since it is not directly related to the key outcomes. $\hat{\alpha}$ is calculated by dividing observed rate by the reported rate.

Our outcomes for known-traffic measurement, and estimated rate measurement may vary depending on the RIPE IPs we used and the traffic data we are using. We tried 5 s of traffic to find out the known traffic rate. This choice is arbitrary, a user can wait for some more time. Given the RIPE IPs that we provided, a user may expect to see 25-50 RIPE queries per second. Please note that, we used a subset of RIPE IPs. A larger RIPE IP set along with their consistent signal would ensure more stable RIPE query rate. We also provided some snapshots for the estimated rate measurement. Please note that, they are just snapshots. Estimated rates are dependent on the observed traffic rates (always varying), and the access fraction.

H.5.2 Building BGP playbook:

We propose a BGP playbook to fight against DDoS attacks. We build the BGP playbook with different routing options and their impacts over traffic distribution (Section 6 and 7 [9]). We show that BGP playbook can help the operators to select the right routing option during an attack event, and a playbook can provide a granular control over traffic distribution.

To reproduce the result, a user needs to announce different BGP configurations, and then run Verfploeter/pinger [3] to learn the prefixes to anycast site catchment. We provide scripts (*playbook_builder* in Peering and *tangler-cli* in Tangled) for our testbeds to makes these announcements automatically [15, 16]. One needs to have access to the testbeds to run this experiment. We used `Peering` [17] and `Tangled` [2] testbeds. These testbeds authorize an anycast prefix for a specific time period. One needs to ask for permission with a proposal to use these testbeds [1, 18]. Our script is dependent on verfploeter/pinger tool which is available online [3], and we provide a binary. This tool needs a target hitlist of IPs which we provided with our dataset (search for internet_address_history_it88w20191127 [6]). We provide a tool named getting_hitlist_ips to parse this raw hitlist file to get the list of responsive IPs. The instruction to run these tools is available in our webpage [15, 16].

To validate our results, we also provide the datasets that we got from our experiments. We include captured pcap files, and data in human-readable format for Peering [12], and in csv format for Tangled [13]. To reproduce results from the collected data, we also provide tools called *load_parser* and *ParsingLoad* in Peering [15], and *measurement scripts* in Tangled [16].

Our result is dependent on the stability of the network state. Since anycast catchment is fairly stable, we expect to get a slight variation but similar results if we rerun the experiment.

Detailed steps: We provide an example here to reproduce Figure 5 from our paper. Other similar graphs and tables like Figure 5—Figue 7, Figure 8, Table 5, Table 6, Figure 9, Figure 10, Table 7, Figure 13, Figure 14 can be generated using the similar process. Please note that figures for community strings and path poisoning (Figure 7, Figure 10, and Figure 13) for `Peering` utilizes only ParsingLoad utility alone (we provide the details later in this subsection).

At first, one needs to run *playbook_builder* tool to make BGP announcements for every prepending option. This step is dependent

on getting access from the Peering testbed. Also, Internet routing changes, and we will not get the same outputs that we received while doing the experiment. As a result, we provide the collected data in pcap form to skip this step. Please find this dataset in peering and root DNS dataset—prepending (3 sites) 2020-02-24. The other datasets for other figures mentioned in prior paragraph are also provided.

To recreate Figure 5, we provide the following datasets:

1. The pcap files in peering and root DNS dataset: anycast_catchment_distribution-20200224/Path_Prepending_AMS,BOS,CNF-20200224,
2. The IP hitlist internet_address_hitlist_it88w-20191127/internet_address_hitlist_it88w-20191127.fsdb.bz2,
3. Some "load" data, provided with the software tool (we consider catchment in this figure so a full load data is not important).

After having these data, one needs to run anygility-peering/src/getting_histlist_ips/getting_hitlist_ips on the hitlist:

bzcat /data/internet_address_hitlist_it88w-20191127/internet_address_hitlist_it88w-20191127.fsdb.bz2 | python3 ./getting_hitlist_ips/data/ip_list_20191127.txt.

This will create a text file, ip_list_20191127.txt, containing one responsive IP address per line.

Then one needs to run anygility-peering/src/load_parser/load_parser.sh on the pcaps with the generated IP hitlist and sample load-file, and its corresponding load-date (e.g. –load=. –ldate=2022-02-01 to use the one provided with the tool):

bash load_parser.sh --numbers=3 --sites=AMS,BOS,CNF --date=2020-02-24 --dir=/data/anycast_catchment_distribution-20200224/Path_Prepending_AMS,BOS,CNF-20200224/ --load=.--ldate=2022-02-01 --hitlist=/data/ip_list_20191127.txt Please note that the trailing / in the –dir argument is necessary.

This will run the ParsingLoad java utility for each announcement configuration, which will

- generate .dat files with ping responses from the .pcap files using pingextract utility.
- compute catchment data, both in terms of /24-blocks and "load" and store these as .txt files inside the data directory. For each announcement configuration, two files <DATE>-catchment-percentage.txt and <DATE>-load-percentage.txt are created. In addition, a combined all-<DATE>-load-<LOAD-DATE>.txt file is created in the data root directory.

The content of all-<DATE>-load-<LOAD-DATE>.txt consists of multiple blocks of this form:

<routing-configuration-path>

- <missing /24 count> <missing /24 relative>

site_1 <site_1 /24 count> <site_1 /24 relative> <site_1 /24 relative received>

[...]

site_n <site_n /24 count> <site_n /24 relative> <site_n /24 relative received>

multiple <multiple /24 count> <multiple /24 relative> <multiple /24 relative received>

- <missing load count> <missing load relative>

site_1 <site_1 load count> <site_1 load relative> <site_1 load relative received>

[...]

site_n <site_n load count> <site_n load relative> <site_n load relative received>

multiple <multiple load count> <multiple load relative> <multiple load relative received>

Figure 5 then shows bar-graphs created from the <site_x /24 relative received> values.

Using ParsingLoad alone: The script load_parser utilizes ParsingLoad for each of the path prepending configurations. When we are not parsing path prepending configurations, we can just utilize ParsingLoad utility alone. We utilize ParsingLoad alone for community strings and path poisoning (Figure 7 and Figure 13). We run ParsingLoad for each of these routing configuration separately.

java -jar ParsingLoad.jar 3 AMS,BOS,CNF anycast_catchment_distribution-20200224/Community_Strings_AMS,BOS,CNF-20200225/2020-02-25-AMS,BOS,CNF-AMS-ALL-PEERS/ /nfs/lander/traces/verfploeter/broot_verfploeter/Peering/Peering_Mapping/2020/community_strings/2020-02-25-AMS,BOS,CNF-AMS-ONLY-PEERS/ 2020-02-25 loads/ 2020-02-22

The output has the same format like all-<DATE>-load-<LOAD-DATE>.txt as we mentioned above. We combine these generated files to build Figure 7 and Figure 13. We use ParsingLoad separately for each routing configuration with community strings and path poisoning. But a script for all the community string and path poisoning options is also possible. For path poisoning, we used poisoning datasets (inside anycast_catchment_distribution-20200224) for AS174 (Tier-1), AS8283 (Transit-2), and AS12859 (Transit-1).

H.5.3 Selection from the playbook:

We provide a tool [14] to select the right routing configuration from the BGP playbook (Section 3.4.2 [9]). Using this tool, we show that an automated approach can be useful to select the right routing approach.

Our selection tool provides output based on the current playbook, and offered load. To show how the selection tool works, we provide a sample playbook (based on Table 5 [9]), and a load file. When the users run the tool with the given inputs, they can see the selection output. We also include a tool named *bgp-tuner* for showing the graphical interface [16].

Depending on the playbook and offered load, one can observe a different output, which can be a complete different policy selection.

Detailed steps: We provided a sample playbook and offered load file with the playbook_tuner tool. Please run the following command to see the outputs from this program:

cat load.txt | ./playbook_tuner –setup "playbook.txt"

This will result the following output:

Overloaded site: AMS

Suggested config: 1AMS, Estimated load distribution: 41292.64 29494.75 41292.64

Other configs: Poison-Tier-1, Estimated load distribution: 41292.64 29494.75 41292.64

Other configs: Poison-Tier-2, Estimated load distribution: 41292.64 29494.75 41292.64

This tells that prepending AMS by 1 would provide the best possible load distribution. Some other options are also possible.

H.5.4 Attack mitigation:

We show that BGP playbook is helpful to mitigate the real-world DDoS events.

To reproduce the same result, we provide the B-root attack traces in pcap and in message question formats [12]. Due to privacy reason,

we cannot share the attack data from the Enterprise and Dutch National Scrubbing Center. We also provide the catchment distribution for different BGP changes [12, 13]. Matching the attack prefixes and attack loads to the prefix-wise catchment gives us the traffic distribution at different sites. If one wants to test the B-root event, they need to run *TimeBasedPrefixLoad* tool to get the per prefix attack load [15]. Then one needs to run *AnycastSiteLoad* program to get the per anycast site load [15].

Since the attack and catchment mapping are fixed, we expect to get the same results that we showed in the paper.

Detailed steps: We show the detailed steps to generate Figure 11(a) here. All other subfigures of Figure 11 and Figure 16 can be generated using the similar process.

To generate Figure 11(a), we need the following datasets:

1. peering and root DNS dataset: B_Root_Anomaly_message_question-20170306/: Figure 11(a) shows 10000 s of traffic. To make the data processing faster, we recommend to use a subset of this whole timeframe. We recommend the user to download the datasets from 06:40:00 AM to 06:50:00 AM to reproduce a fraction of the whole timeframe combining both attack and non-attack period. The file names represent the dates and times (format: YYYYMMDD-HHMMSS-*).
2. peering and root DNS dataset: anycast_catchment_distribution-20200224/Path_Prepending_AMS,BOS,CNF-20200224/2020-02-24-AMS,BOS,CNF/
3. peering and root DNS dataset: /anycast_catchment_distribution-20200224/Community_Strings_AMS,BOS,CNF-20200225/2020-02-25-AMS,BOS,CNF-AMS-Transit-1-Trial-2/(update: this Trial-2 dataset is newly added. We also provided Trial-1 dataset for 2020-02-25-AMS,BOS,CNF-AMS-Transit-1 which will give a similar output, but we did not use that in the paper).

At first, run the TimeBasedPrefixLoad java utility on the downloaded message_question format data. We only need time, source IP and message length for our measurement. message_question formatted files have several attributes/columns. We used fsdb tool to retrieve the times, source IPs, and message length [4]. Please follow the instruction to install FSDB from here: https://www.isi.edu/~johnh/SOFTWARE/FSDB/perl-Fsdb-2.74_README.html. Next, use the following command to run TimeBasedPrefixLoad jar to generate the prefix-wise load for each 5 s:

xzcat B_Root_Anomaly_message_question-20170306/06/20170306-044* | dbcol time srcip msglen | java -jar TimeBasedPrefixLoad.java output-20170306/ 192.228.79.64,2001:500:84::bb26:87a2.

Here, dbcol is a utility from FSDB to select the right column from the message_question format dataset. output-20170306 will have multiple txt files named with a number indicating the time segment. This command will generate prefix-wise load at every 5 s in output-20170306 directory: <network_prefix> <number_load> <bytes>.

Then we run AnycastSiteLoad java utility to find out the per site load at every 5 s. We run this utility for two routing configurations—one without any routing change and one with announcing only to Transit-1.

java -jar AnycastSiteLoad.jar 3 AMS,BOS,CNF anycast_catchment_distribution-20200224/Path_Prepending_AMS,BOS,CNF-20200224/2020-02-24-AMS,BOS,CNF/ 2020-02-24 output-20170306/ 2017-03-06,

java -jar AnycastSiteLoad.jar 3 AMS,BOS,CNF anycast_catchment_distribution-20200224/Community_Strings_AMS,BOS,CNF-20200225/2020-02-25-AMS,BOS,CNF-AMS-Transit-1-Trial-2/ 2020-02-25 output-20170306/ 2017-03-06.

Please note that these two commands utilize output-20170306 that we generated in our previous step. These two commands generate two files in the corresponding catchment directory named as <CATCHMENT-DATE>-load-<ATTCK-DATE>-ingress.txt. The output format inside the file: <time> <site-1> <count-site-1> <bit-site-1> <...> <site-n> <count-site-n> <bit-site-n>. The first file contains load without any routing change, the second file contains load after announcing only to Transit-1. We combine these two files to show non-attack period (no policy deployed), and period when the route propagation is done (when we deployed Transit-1).

To match the results with the Figure 11(a), the first output file will contain (<count-site-n> column) traffic load during normal period (before 0 s from the graph with around 20k packets/s). The first output file also contains the attack traffic (AMS load over 60k packets/s after 160 s of the first file). This is similar to the traffic from 0 s to 300 s of Figure 11(a). After that we announce only to Transit-1 (after 300 s of Figure 11(a)). The second output file contains this data (after 160 s from the file).

H.6 Notes

Considering the real datasets are big, and time expensive to run, we include smaller datasets collected using a small hitlist fraction (0.1% of original size) in experiments with Tangled. While the produced playbook will differ from paper results, we believe it can help for testing purpose. For Peering tools, we sometimes include smaller sample supporting data files.

If desired, we can provide access to the Tangled testbed. Access to Peering testbed is dependent on the approval from Peering admins.

H.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

References

[1] Tangled admins. Tangled anycast testbed. https://anycast-testbed.nl/, 2019. [Online; accessed 15-Feb-2022].

[2] Leandro M Bertholdo, Joao M Ceron, Wouter B de Vries, Ricardo de Oliveira Schmidt, Lisandro Zambenedetti Granville, Roland van Rijswijk-Deij, and Aiko Pras. Tangled: A cooperative anycast testbed. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 766–771. IEEE, 2021.

[3] Wouter De Vries. Verfploeter/pinger: Active measurement of anycast catchements. https://ant.isi.edu/software/verfploeter/pinger/index.html, 2019. [Online; accessed 15-Feb-2022].

[4] John Heidemann. John heidemann / software / fsdb. https://www.isi.edu/~johnh/SOFTWARE/FSDB//, 1991. [Online; accessed 19-Mar-2022].

[5] Analysis of Network Traffic (ANT) group. Ant dataset requests. https://ant.isi.edu/datasets/requests.html, 2022. [Online; accessed 15-Feb-2022].

[6] Analysis of Network Traffic (ANT) group. Ant datasets. https://ant.isi.edu/datasets/index.html, 2022. [Online; accessed 15-Feb-2022].

[7] Root Server Operators. Events of 2015-11-30. https://root-servers.org/media/news/events-of-20151130.txt, 2015. [Online; accessed 12-Oct-2021].

[8] Root Server Operators. Events of 2016-06-25. https://root-servers.org/media/news/events-of-20160625.txt, 2016. [Online; accessed 12-Oct-2021].

[9] A S M Rizvi, Leandro Bertholdo, João Ceron, and John Heidemann. Anycast agility: Network playbooks to fight DDoS. In *Proceedings of the 31st USENIX Security Symposium*, page to appear. USENIX, August 2022.

[10] A S M Rizvi, Leandro Bertholdo, Joao Ceron, and John Heidemann. anygility - anycast agility tools: playbook builder and decision maker. https://ant.isi.edu/software/anygility/index.html, 2022. [Online; accessed 2-Mar-2022].

[11] A S M Rizvi, Leandro Bertholdo, Joao Ceron, and John Heidemann. Artifacts about anycast agility against ddos. https://ant.isi.edu/datasets/anycast/anycast_against_ddos/index.html, 2022. [Online; accessed 2-Mar-2022].

[12] A S M Rizvi, Leandro Bertholdo, Joao Ceron, and John Heidemann. Datasets about anycast agility against ddos in peering testbed. https://ant.isi.edu/datasets/anycast/anycast_against_ddos/peering/index.html, 2022. [Online; accessed 2-Mar-2022].

[13] A S M Rizvi, Leandro Bertholdo, Joao Ceron, and John Heidemann. Datasets about anycast agility against ddos in tangled testbed. https://ant.isi.edu/datasets/anycast/anycast_against_ddos/tangled/index.html, 2022. [Online; accessed 15-Feb-2022].

[14] A S M Rizvi, Leandro Bertholdo, Joao Ceron, and John Heidemann. Tools about anycast agility against ddos. https://ant.isi.edu/software/anygility/system/index.html, 2022. [Online; accessed 2-Mar-2022].

[15] A S M Rizvi, Leandro Bertholdo, Joao Ceron, and John Heidemann. Tools about anycast agility against ddos in peering testbed. https://ant.isi.edu/software/anygility/peering/index.html, 2022. [Online; accessed 2-Mar-2022].

[16] A S M Rizvi, Leandro Bertholdo, Joao Ceron, and John Heidemann. Tools about anycast agility against ddos in tangled testbed. https://ant.isi.edu/software/anygility/tangled/index.html, 2022. [Online; accessed 2-Mar-2022].

[17] Brandon Schlinker, Todd Arnold, Italo Cunha, and Ethan Katz-Bassett. PEERING: Virtualizing BGP at the Edge for Research. In *Proc. ACM CoNEXT*, Orlando, FL, December 2019.

[18] Peering The BGP Testbed. Peering the bgp testbed. https://peering.ee.columbia.edu/, 2019. [Online; accessed 15-Feb-2022].





A Artifact Appendix

A.1 Abstract

Our artifact includes the regexps processed in this paper, a record of the NPM packages analyzed, the REGULATOR workflow and source code, the results produced by both our tool and those used for comparison, and software for computing values and figures found in this paper.

We provide a x86-64 docker container with all prerequisites necessary to compile REGULATOR. A pre-compiled version is also included.

Our results can be validated by re-running the tool to detect and verify ReDoS-vulnerable regexps.

A.2 Artifact check-list (meta-information)

- **Data set:** We use two different datasets in our paper. The first (called Base Dataset in our paper) was sourced from three different collections used in previous ReDoS research (known as Corpus, RegexLib and Snort). The second (called NPM dataset) was instead created during our research, by scraping and extracting the regular expression used in the 10,000 most popular NPM packages. Both datasets are included in the docker container under `/artifacts/data/regex.csv`.
- **Run-time environment:** The software is evaluated using Python 3.8, NodeJS 10.19.0, Postgresql 12, and gcc 9.3.0, running on Ubuntu 20.04 in Docker 20.10.7.
- **Run-time state:** Regulator is based on fuzzing, so results might slightly differ between each run.
- **Execution:** Since Regulator fuzzes each regular expression for a given amount of time, we recommend to run the tool on a machine with no significant background tasks.
- **Metrics:** Each tool evaluated in our paper reports whether a regular expression is vulnerable to ReDoS. Our artifact reports the following metrics: true positives, false positive and false negatives of each tools (Table 3 and Table 4 of our paper).
- **Output:** For each regular expression, a record is produced of the fuzz witness, the classified growth-function (if super-linear), whether it was verified to cause significant slow-down, and the minimum string-length length required to achieve that slow-down.
- **Experiments:** Included are setups for these experiments: running REGULATOR against regular expressions, and running the comparison tools REGULATOR-PerfFuzz, ReScue, NFAA, Revealer, RXXR2, and Rexploiter against regular expressions.
- **How much disk space required (approximately)?:** Approximately 10 GB of disk space is required.
- **How much time is needed to prepare workflow (approximately)?:** About 30 minutes is required to laod the docker container, start services, and queue regexps in the workflow.
- **How much time is needed to complete experiments (approximately)?:** The experiments require approximately 10,000 CPU-hours. The workflow can be configured to make use of multiple CPU cores at once, to reduce the wall-clock time required.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** 10.5281/zenodo.5669243

A.3 Description

A.3.1 How to access

The docker container where all artifacts are stored can be downloaded at the following url: https://doi.org/10.5281/zenodo.5669243

A.3.2 Hardware dependencies

Regulator does not have any particular hardware dependency (the docker container was tested on a x86-64 Linux system), but we recommend to using a server with a substantial number of CPU cores.

A.3.3 Software dependencies

The only software dependency is `docker` (tested on version 20.10.7), plus a `gzip` decompression utility.

A.4 Installation

The artifact contains a docker container. After downloading the compressed image `regulator_artifacts.tar.gz`, decompress it using a decompression tool. For example, on most Linux systems: `gzip -d regulator_artifacts.tar.gz` to produce the file `regulator_artifacts.tar`. It can be loaded into docker with the following command `docker load < regulator_artifacts.tar`. Then, the container can be started with `docker run -it regulator_artifacts /bin/bash`.

A.5 Experiment workflow

The core results from our paper can be reproduced by running REGULATOR and previous research tools against the Base and NPM dataset. The previous tools tested in this paper are: RXXR2, Rexploiter, NFAA, ReScue, PerfFuzz, Revealer. The first four tools were packaged by Davis Jamie in the `vuln-regex-detector` project [1]. In the following sections we therefore present 4 different workflows to run REGULATOR, PerfFuzz, Revealer, and vuln-regex-detector.

REGULATOR. The experiment workflow to run REGULATOR is documented in `/artifacts/detectors/regulator/README.md`. Before running our tool, the regular expressions must be loaded inside a postgres database using the `add_to_queue.py` script.

REGULATOR has then three phases:

[1] https://github.com/davisjam/vuln-regex-detector

1. Fuzzing Stage: the target regular expression is fuzzed. The output of this step is a witness string, i.e. the string that has the highest number of executed bytecode instructions. This step is implemented in the `fuzz_from_queue.py` script.

2. Pumping Stage: the witness string is translated in a pump formula. This step is implemented in the `pump_all.py` script.

3. Dynamic Validation: the pump formula is tested against the irregexp engine. If the formula causes a slowdown of more than 10 seconds then the target regular expression is marked as vulnerable to ReDoS. This step is implemented in the `binsearch_pump.py` script.

PerfFuzz. The workflow to run PerfFuzz is quite similar to REGULATOR's, and more instructions can be found under `/artifacts/detectors/regulator/README.md`.

Revealer. The workflow to run Revealer is documented in `/artifacts/detectors/revealer/README.regulator.md`. To run this tool, invoke the script `run_with_timeout.py`.

vuln-regex-detector. The workflow to run ReScue, RXXR2, NFAA, and Rexploiter. This is documented in `/artifacts/detectors/davis-detectors/README.md`.

A.6 Evaluation and expected results

In this paper we show that REGULATOR outperforms previous ReDoS detectors. This is the core result of this research, and is shown in Table 3 and Table 4 of the paper. Rerunning the tool should show more true positive detections than prior work.

There are two ways to reproduce these results. The first is to re-use the output of our experiments (stored under `artifacts/data/`, the second is to run the workflows discussed in the previous section and copy these newly generated results into `/artifacts/data`. See the `README` for each workflow for more details.

In both cases, the script `/artifacts/scripts/analyze_results.py` will summarize the results and produce the numbers contained in Table 3 and Table 4.

If running the entire workflow requires too many resources, REGULATOR can be more quickly evaluated by running the workflow for a random subset of regexps which were reported as vulnerable in this work. We expect a high percentage (at least 80%) to be reproducible.



A Artifact Appendix

A.1 Abstract

The artifact is an implementation and empirical evaluation of Aardvark, an authenticated dictionary.

The artifact contains two sets of benchmarks for evaluation in the paper. First, it contains microbenchmarks of vector commitment operations which compare those used in the paper with those in EDRAX (a related system), and with a basic Merkle Tree. Second, it contains a benchmark of the dictionary operations themselves from the perspective of both a validator and an archive, with the dictionary integrated into the backend of the Algorand cryptocurrency. The objective of these benchmarks is to substantiate the paper's claims of computational efficiency, which is difficult to analytically evaluate. In particular, these benchmarks measure the latency of key vector commitment and dictionary operations.

The artifact may be validated by downloading it from the public GitHub repository URL provided and running the evaluation scripts, which are part of the repository. The expected result of artifact evaluation is that the latency measurements match those in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Authenticated dictionary
- **Program:** Custom benchmarks, included
- **Compilation:** g++ 9.3.0, rustc 1.54.0-nightly (126561cb3 2021-05-24), go 1.16.4
- **Metrics:** Latency
- **Output:** File, measured characteristics, expected result included
- **Experiments:** OS Scripts
- **How much disk space required (approximately)?:** 1GB
- **How much time is needed to prepare workflow (approximately)?:** 6hrs
- **How much time is needed to complete experiments (approximately)?:** 13hrs
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT, GPLv3
- **Archived (provide DOI)?:** Yes, https://github.com/derbear/aardvark-prototype/tree/dd8f6aaf5f76173118f3f3decbe099bda5972ce2

A.3 Description

A.3.1 How to access

Clone the repository and its submodules from GitHub at the following URL: `https://github.com/derbear/aardvark-prototype/tree/dd8f6aaf5f76173118f3f3decbe099bda5972ce2`. For instance, run

```
git clone --recurse-submodules \
https://github.com/derbear/aardvark-prototype.git
git checkout dd8f6aaf5f76173118f3f3decbe099bda5972ce2
```

EDRAX and its dependencies are under the `edrax` subdirectory, the implementations of vector commitments and Merkle trees are under the `veccom-rust` subdirectory (which depends on the `pairing-fork` subdirectory), and the Algorand implementation resides in the `go-algorand` subdirectory with the Aardvark implementation in `go-algorand/ledger`.

The `--recurse-submodules` option initializes the repositories to their correct versions. The commits corresponding to this document's version of the artifact for the top-level repository, `veccom-rust`, and `edrax` are all additionally labelled `usenix22-artifact` through `git tag`. To confirm that the versions of all submodules are correct, run `git submodule status --recursive` from `aardvark-prototype`, which should produce the following hashes.

```
1f1a3748d1530da1e75fadbce987ee6e6fa3fd1d edrax
530223d7502e95f6141be19addf1e24d27a14d50
 edrax/ate-pairing
a34850b2df66a186c8d947b4d72acc839926321f edrax/xbyak
cff079d3f78daa48d25183292960c21da9cdf152 pairing-fork
d72ed3c8b0e4624053360591fcc8d03ce720ae90 veccom-rust
```

If you did not supply the `--recurse-submodules` option above, you can alternatively initialize these submodules by running the following command from `aardvark-prototype`.

```
git submodule update --init --recursive
```

A.3.2 Hardware dependencies

To reproduce results regarding the authenticated dictionary's scalability, 32 cores are required. The provided benchmarking script in the repository assumes the presence of at least 64 cores.

Around 110MB of disk space is required to clone the entire git repository. Around 1GB of disk space is required to run the experiments.

A.3.3 Software dependencies

Building the software depends on the compilers `g++ 9.3.0`, `rustc nightly-2021-05-25`, and `go 1.16.4`; on the `libgmp3` library; and on the build tools `cmake`, `make`, `autoconf`, `automake`, and `libtool`. Running benchmarks depends on `numactl`.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, Privacy, and Ethical Concerns

N/A

A.4 Installation

The following instructions assume that your working directory is `$TOP` and that you are running Ubuntu 18.04 or 20.04. (Older versions of Ubuntu may require modifying these steps.)

A.4.1 Obtaining the source code

```
git clone --recurse-submodules \
https://github.com/derbear/aardvark-prototype.git
git checkout dd8f6aaf5f76173118f3f3decbe099bda5972ce2
git submodule update --init --recursive
```

A.4.2 Installing dependencies for EDRAX

```
sudo apt update
sudo apt install cmake g++ libgmp3-dev

# ignore errors while building dependencies here
cd $TOP/aardvark-prototype/edrax/ate-pairing ; make
cd $TOP/aardvark-prototype/edrax/xbyak ; make

cd $TOP/aardvark-prototype/edrax ; cmake . && make
```

A.4.3 Installing dependencies for vector commitments

```
# install rustup
curl --proto '=https' --tlsv1.2 \
-sSf https://sh.rustup.rs | sh
# input 1 for standard installation

# add to shell profile for this to be persistent
source $HOME/.cargo/env

rustup install nightly-2021-05-25
rustup default \
nightly-2021-05-25-x86_64-unknown-linux-gnu

cd $TOP/aardvark-prototype/veccom-rust ;
cargo build --release
```

A.4.4 Installing dependencies for Aardvark, integrated into Algorand

```
sudo apt update
sudo apt install autoconf automake libtool numactl

wget https://golang.org/dl/go1.16.4.linux-amd64.tar.gz
tar -C $TOP -xzf go1.16.4.linux-amd64.tar.gz

# add to shell profile for this to be persistent
export PATH=$PATH:$TOP/go/bin
export GOPATH=$TOP/go

cd $TOP/aardvark-prototype/veccom-rust ;
cargo build --release
cd $TOP/aardvark-prototype/go-algorand ; make
# input N when prompted, and ignore Makefile error
```

A.5 Experiment workflow

The following instructions assume that your working directory is `$TOP`.

A.5.1 EDRAX microbenchmark

The EDRAX binary resulting from compiling calls into the EDRAX implementation. It executes 100 iterations to warm up the machine state and then performs 1000 measurements of the implemented Verify, CommitUpdate, and ProofUpdate operations. The script `edrax/bench.sh` invokes the binary with the argument 10, which corresponds to vectors with size 1024, and writes the results as a CSV file to the file `bench.csv` to the current directory.

A.5.2 Aardvark vector commitment microbenchmark

The binary resulting from compiling `veccom-rust/src/bin/run_aardvark_bench.rs` calls into the implementation of vector commitments for Aardvark, as well as an implementation of a Merkle Tree. It executes 100 iterations to warm up the machine state and then performs the passed-in number of measurements of the operations described in §4.1. The script `veccom-rust/bench.sh` invokes the binary with the argument corresponding to vectors with size 1024 and with 1000 iterations, and it writes the results as a CSV file to the file `bench-results.txt` to the current directory.

A.5.3 Aardvark dictionary benchmark

Aardvark is implemented as a modification of the database of the Algorand cryptocurrency and is contained inside the repository under the subdirectory `go-algorand/ledger`. The benchmark itself is written as a Go test within the file `perf_test.go`, and it consists of a workload generation program (written as a Go test `TestWorkloadGen` for convenience), as well as timed benchmarks (written as a Go test `TestTimeWorkload`).

To generate the workload (which takes roughly 5 hours on the paper hardware), run the following:

```
cd $TOP/aardvark-prototype/go-algorand/ledger ;
./bench.sh
```

This will create in the `ledger` subdirectory the files `workload-{init-}{c,d,m}`, which correspond to the initialization data and sample load transactions for creation, deletion, and modification benchmarks, respectively. Once the workloads are created, the benchmarks may be run against them.

Note that if you are executing these commands over an SSH connection, a dropped connection will terminate the generation process, and you will need to reissue the command from the beginning. We suggest using commands such as `nohup`, `screen`, or `tmux` to prevent a dropped connection from interrupting the command.

A.6 Evaluation and expected results

The paper claims that Aardvark is a secure authenticated dictionary with substantial storage savings and short proofs, and it can process more than a thousand operations per second.

The security of Aardvark is justified through a paper proof. The evaluation contains an analysis of the storage savings and proof sizes,

which are straightforward to compute. The rest of the evaluation performs an empirical analysis to obtain the throughput of a prototype implementation of Aardvark, which is shown in the artifact.

The paper obtains the following empirical results in the evaluation.

1. While Aardvark's vector commitments are more computationally intensive than Merkle trees, their costs are similar to those in EDRAX without use of a SNARK.
2. A 32-core Aardvark validator can process 1–3 thousand operations per second. Validator costs benefit from parallelization.
3. Costs for archives are reasonable: each core can process about 10 deletion operations per second or 20 modification/insertion operations per second.

The concrete numerical results are displayed on Tables 1 and 4 as well as Figures 3 and 4 in §8. Raw expected results for vector commitment microbenchmarks are in `edrax/results` and `veccom-rust/bench-results`, while raw expected results for validator and archive operations are in `go-algorand/ledger/validators.csv` and `go-algorand/ledger/archives.csv` respectively.

The following instructions assume that your working directory is `$TOP`.

A.6.1 Microbenchmarks

The paper claims in §8.1, Table 1 concrete latency numbers for key vector commitment operations for EDRAX, our implementation of Aardvark, and our implementation of a basic Merkle Tree. Reproduce them as follows.

```
# benchmark EDRAX latency
cd $TOP/aardvark-prototype/edrax ; ./bench.sh
# time ./bench.sh takes <1min on paper's hardware

# benchmark vector commitments latency
cd $TOP/aardvark-prototype/veccom-rust ; ./bench.sh
# time ./bench.sh takes <3mins on paper's hardware
```

The output results for EDRAX are in `edrax/bench.csv`, while the expected raw results in the paper are in `edrax/results`. The output results for the vector commitments are in `veccom-rust/bench-results.txt`, while the expected raw results in the paper are in `veccom-rust/bench-results`.

A.6.2 Validator and Archive throughput

The paper claims in concrete latency measurements for insertion, modification, and deletion operations for our implementation of Aardvark for validators (§8.3, Table 4 and Figure 3) and for archives (§8.4, Figure 4). Reproduce them as follows.

```
# first, generate the workload as described in the
# previous section

# runs 3 scaling tests on validators
cd $TOP/aardvark-prototype/go-algorand/ledger ;
./cores.sh
# time ./cores.sh takes <4hrs on paper's hardware

# runs 3 tests on archives
cd $TOP/aardvark-prototype/go-algorand/ledger ;
./acores.sh
# time ./acores.sh takes <4hrs on paper's hardware
```

The results for validators are in files named `outN.txt`, where N is the number of cores and is either 1, 2, 4, 8, 16, or 32, while the results for archives are in a file named `aout1.txt`. By default, both of these tests run 3 trials each. Expected raw values for these results for 10 total trials each, manually merged, are in `go-algorand/ledger/validators.csv` and `go-algorand/ledger/archives.csv` respectively.

Note that if you are executing these commands over an SSH connection, a dropped connection will terminate the experiment process, and you will need to reissue the command from the beginning. We suggest using commands such as `nohup`, `screen`, or `tmux` to prevent a dropped connection from interrupting the command.

A.7 Experiment customization

Different vector sizes may be passed to the vector commitments libraries by modifying the command-line arguments which the `bench.sh` files pass to the binaries.

Modifying `go-algorand/ledger/perf_test.go` will allow modifying the number of initial accounts, the number of load transactions, the number of blocks, and other parameters input to Aardvark. (Modifying any variables here will require regeneration of the workload.)



A Artifact Appendix

A.1 Abstract

The purpose of this artifact is to allow reproduction of the performance results in Section 8, specifically the channel opening microbenchmark table (Figure 6) and the full proof generation benchmark for the case studies (Figure 7). All runtime estimates in this abstract are for a Linux system with an 8-core 2.2 GHz AMD EPYC 7571 CPU and 32 GB of RAM. All of our code is available on GitHub.

After installing dependencies, which should take roughly ten minutes, reproducing these results has two steps: (1) circuit generation and (2) proof generation. Circuit generation takes as input a (roughly) human-readable programmatic description of our circuits written in an extension of Java, and outputs a gate-level description of the corresponding arithmetic circuit. This programmatic circuit description is an intermediate representation obtained by partially compiling the original handwritten xJsnark source code. We do not require the original xJsnark source for the artifact evaluation—reading it requires installing a specific version of a large and unwieldy IDE called MPS—but our GitHub repository includes instructions on viewing the xJsnark source.

Circuit generation involves heavily optimizing the circuit description, and so is computationally quite expensive, and will take up to twenty minutes (to generate the nine example circuits in this artifact). The purpose of re-running the circuit generation as part of the artifact is to allow users to reproduce the claimed gate counts for our circuits. We provide a single script to automatically perform all of circuit generation.

After the circuits' descriptions have been generated, the last step is proof generation. Proof generation takes as input the circuit descriptions as well as sample circuit inputs (e.g., TLS handshake transcripts and ciphertexts), generates public parameters, produces proofs, and verifies them. The provided proof generation script outputs information about the time taken to generate and verify proofs, as well as the sizes of the public parameters. We estimate this will take in total up to twenty minutes (to complete all nine circuits in the artifact).

A.2 Artifact check-list (meta-information)

- **Algorithm:** zkSNARKs, Groth16
- **Program:** xJsnark, libsnark
- **Compilation:** Java, cmake
- **Data set:** Manually generated test data. Included.
- **Run-time environment:** Ubuntu 20.04, OpenJDK 11.0.13
- **Hardware:** 32 GB RAM, 8 cores
- **Metrics:** Circuit size, proving time, verification time, parameter size
- **Experiments:** Bash scripts
- **How much disk space required (approximately)?:** 3 GB
- **How much time is needed to prepare workflow (approximately)?:** 10 min
- **How much time is needed to complete experiments (approximately)?:** 40 min
- **Publicly available:** GitHub: https://github.com/pag-crypto/zkmbs/
- **Archived (stable URL):** https://github.com/pag-crypto/zkmbs/tree/096ed18772d8e63f4a03e7f4d16e118aa3923135

A.3 Description

A.3.1 How to access

Our artifact's code is publicly available on GitHub here:
https://github.com/pag-crypto/zkmbs
This appendix contains all the instructions specific to installation and reproducing the paper's benchmarks.

A.3.2 Hardware dependencies

We recommend using a machine with 8 cores and at least 32 GB RAM.

A.3.3 Software dependencies

The only major dependency is Java. We recommend using a GNU/Linux system and have provided installation scripts compatible with the Ubuntu 20.04 Linux distribution.

A.4 Installation

1. Clone the git repository and change to the root directory (time required: < 1 minute):

```
$ git clone https://github.com/pag-crypto/zkmbs.git
$ cd zkmbs/
```

2. Install jsnark (a library used by xJsnark) and its dependencies by running this script inside `zkmbs/` (time required: 5–10 minutes): `$ ./install_deps_jsnark`

   - If you can't use the script, follow the "jsnark installation instructions" here: https://github.com/akosba/jsnark#prerequisites
   - On some systems, this step may fail when trying to install the dependencies of libsnark as specified in this file: https://github.com/akosba/libsnark/blob/213547311d16644bde7ef806b77dfae25c7f734c/.gitmodules. Please edit all URLs in your local version of the file at `zkmbs/jsnark/libsnark/.gitmodules` (which should be cloned by this point) to use `https` (and not `git`) and try again.

3. Enter gen/ and compile xJsnark: `$ cd gen/` and `$ ./compile_circuits`. The exact output will depend on the system but it should finish without any errors. On Ubuntu, our output looks like this:

```
Note: Some input files use ...
Note: Recompile with -Xlint:unchecked ...
compilation SUCCESS
```

A.5 Experiment workflow

After installation, the structure of the main directories should look like this:

```
zkmbs
    +-- gen
    |    +-- circuits
    |    +-- logs
    |    +-- src
    +-- jsnark
```

The experiment scripts will be run inside gen/. The Java source code describing the circuits is located in gen/src/.

Experiment 1 will generate full circuits from these descriptions and store them in gen/circuits/. Experiment 2 will use these circuit descriptions to generate public parameters and measure proving and verification times using sample input files located in gen/.

A.6 Evaluation and expected results

The main performance claims in our paper are stated in Figures 6 and 7. There are nine circuits involved in our experiments (the five entries of Figure 6 and the four entries of Figure 7). The first experiment reproduces the "Total" columns of the two tables. The second experiment reproduces the "Time" and "SRS" columns while ensuring that verification time is under 5 ms. We recommend using a system with at least 32 GB RAM as generating proofs for the largest of our circuits (ChannelBaseline) requires a lot of heap space, and in fact causes errors on systems with just 16 GB memory.

Note that both Figures 6 and 7 list per-subcircuit gate counts that sum to the "Total" count. Our code only allows verifying the gate counts of the entire circuit, as the per-subcircuit counts were approximated by manually inspecting the functions used to build each circuit.

Experiment 1: Reproduce Gate Counts: The aim is to generate circuits from our descriptions and reproduce the total gate counts of each circuit (the **Total** columns of the two tables). This experiment can be repeated by running the script `./reproduce_total_counts` (time required: 20 minutes) in the gen/ directory. The script outputs into file column_total.txt, which should look like this after the script finishes:

```
ChannelBaseline              747.9   # BCO
ChannelShortcut              111.1   # SCO
Channel0RTT                   60.7   # ECO
ChannelAmortized              19.1   # ACO^AES
ChannelAmortized_ChaCha        8.7   # ACO^Cha
Firewall_HS                  150.1   # Firewall
DNS_Amortized_ChaCha          17.6   # DoT
DNS_Amortized_doh_get         48.1   # DoH GET
ODOH_Amortized                48.1   # ODoH
```

Note that the "# ..." are added here to map to the abbreviations used in Figures 6 and 7. The numbers obtained should be very close to the ones above with perhaps slight variation coming from the performance of xJsnark's optimizer on different systems. Some of the values shown here are different than that of the "Total" columns in Figures 6 and 7 as those were rounded for presentation.

Experiment 2: Reproduce Times and SRS: The aim is to reproduce the structured reference string sizes (**SRS** columns), proving (**Time** columns) and verification time (always under 5 ms) for each circuit. This experiment can be repeated by running the script `./reproduce_times_srs` (time required: 20 minutes) inside the gen/ directory.

The script outputs into file columns_ptime_srs_vtime.txt, the contents of which after a sample execution are as follows:

```
ChannelBaseline            92.7 s   1179 MB    2.6 ms
ChannelShortcut            15.6 s    148 MB    1.6 ms
Channel0RTT                 8.4 s     79 MB    1.6 ms
ChannelAmortized            2.9 s     26 MB    1.7 ms
ChannelAmortized_ChaCha     1.4 s     13 MB    1.6 ms
Firewall_HS                21.2 s    206 MB    1.6 ms
DNS_Amortized_ChaCha        3.1 s     29 MB    2.1 ms
DNS_Amortized_doh_get       6.8 s     72 MB    2.6 ms
ODOH_Amortized              7.9 s     76 MB    2.6 ms
```

Proof generation is a randomized algorithm; the results reported in the paper are the median of five runs. We have observed variations of up to 15% for proving time and 2 ms for verifier time, in either direction. The script above performs just one run per circuit.

A.7 Experiment customization

We provide two additional scripts to reproduce the above benchmarks for an individual circuit: `./generate_circuit DNS_Amortized_ChaCha` and `./prove_and_verify DNS_Amortized_ChaCha`, where "DNS_Amortized_ChaCha" can be replaced with any of the nine circuits.

A.8 Notes

Custom Inputs. As the circuit metrics we evaluate (gate counts, parameters sizes and running times) are independent of the actual input used to generate the proofs, input customization isn't required to reproduce our results. The experiments generate valid proofs using fixed input files (`test.txt, test_doh.txt, test_wildcard.txt`) provided in the `gen/` directory. These files contain sample data extracted from a real TLS 1.3 connection and a Merkle tree blocklist of two million entries. We provide instructions in our GitHub repository on generating sample data from new DNS requests and custom Merkle trees.

Editing Circuit Descriptions. Our experiments generate circuits using the Java files in the `gen/src/` directory. These are in turn generated from xJsnark's custom language files that are editable only with an IDE called MPS. To inspect and edit our circuits, we recommend installing the MPS IDE by following the instructions here in our GitHub repository: https://github.com/pag-crypto/zkmbs#installation-instructions-mps.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

We present our Otti USENIX '22 artifact. It is a docker container that orchestrates the components of Otti to a single interface. To build the docker container and execute the script that reproduces our results, see README.md in our repository eniac/otti. The docker container is composed of 1. the Otti compiler from eniac/otti (Note: Otti was built on top of the Haskell CirC compiler, and later ported to the Rust CirC compiler. Both are included.) 2. The Spartan zkSNARK backend from microsoft/Spartan 3. The compatibility interface between compiler and Spartan in elefthei/spartan-zkinterface. We also fetch their dependencies, which are broadly Haskell, Python, and Rust's build tools, the lpsolve CLI, csdp, scikit-learn, the flatbuffer library, the Z3 model checker and more small, standard libraries in Haskell and Rust.

We also include in our repository representative datasets of linear programming (LP) [1], semi-definite programming (SDP) [3], and the datasets for stochastic-gradient descent (SGD), accessible by installing the PMLB [4] Python library. Our docker container includes scripts to run Otti end-to-end – generate C files from datasets, execute and compile C files to R1CS, and finally prove and verify their correct execution with Spartan.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Otti uses optimization certificates to produce nondetermanistic checkers for zkSNARKS, as detailed in the paper.
- **Compilation:** Otti has a compiler which is included in the container.
- **Transformations:** Otti has transformations from model files for LP, SDP, SGD to C files which are also included as Python scripts.
- **Binary:** Binaries for LP solve [2] for x86_64 UNIX machines are included in the container. As we are not certain regarding compatibility with Apple M1, we would recommend running the container on a x86_64 architecture.
- **Data set:** We use the NETLIB [1], SDPLIB 1.2 [3], and PMLB [4] datasets which are publicly available and relatively small – in the order of a few MB. Representative examples from these datasets are included in the repository and you can refer to our results in the paper for the complete list.
- **Run-time environment:** Docker community edition is required, platform independent.
- **Hardware:** For running large datasets, a computer with > 256GB RAM is required. Small datasets can be run on personal computers.
- **Run-time state:** No
- **Execution:** Execution time varies from small to large datasets and the available memory in the machine. Small ones are really fast and finish in a few minutes but larger ones can take hours.
- **Security, privacy, and ethical concerns:** No
- **Metrics:** Execution time, prover time, verifier time, proof size, number of constraints.
- **Output:** Our result is a total runtime measurement and a "Verification Successful" message that confirms end-to-end execution was proven to the verifier
- **Experiments:** Docker container takes care of setup. Variation should be small (5-10%) in runtimes depending on the machine. Variation in constraint and proof sizes should be 0.
- **How much disk space required (approximately)?:** The docker container requires a substantial amount of disk space, between 20GB-30GB.
- **How much time is needed to prepare workflow (approximately)?:** The docker container builds in about an hour.
- **How much time is needed to complete experiments (approximately)?:** Smaller examples can be run immediately and take a couple of minutes, larger examples must be downloaded, but should not take more than an hour or so.
- **Publicly available (explicitly provide evolving version reference)?:** https://github.com/eniac/otti
- **Code licenses (if publicly available)?:** MIT license
- **Data licenses (if publicly available)?:** [1, 3] are very old and no licensing information was found, [4] is under MIT license.
- **Workflow frameworks used?:** No
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/eniac/otti/releases/tag/v1.0

A.3 Description

How to access Clone repository from GitHub: https://github.com/eniac/otti/releases/tag/v1.0

Hardware dependencies X86_64 machine with a sufficient amount of RAM memory (> 200GB) if evaluating large datasets.

Software dependencies Docker community, latest version.

Data sets See [1, 3, 4].

Models N/A

Security, privacy, and ethical concerns N/A

A.4 Installation

Cloning To clone the repository and its submodules run `git clone -recursive https://github.com/eniac/otti.git`

Building First, make sure you have installed Docker CE: https://docs.docker.com/get-docker/ Then build the Otti container: `docker build -t otti .` Then run the container with 200GB of memory and get terminal access: `docker run -m 200g -it otti`

Reproducing experimental results After connecting to the Docker container, run the following script to reproduce the experimental results from Otti: `./run.py [-lp | -sdp | -sgd] [-small | -full | -custom datasets/<path to dataset>]`

One of the `-lp | -sdp | -sgd` options is required. Then either execute with the `-small` or `-full` flag, or the `-custom` flag with an explicit path to a dataset file.

Running the small suite A subset of each dataset that can be reproduced on a personal computer with x86_64 architecture and >= 12GB of RAM. These datasets are expected to take less than 1 hour.

Running the full suite A subset of each MPS dataset that can be reproduced on a large machine with x86_64 architecture and > 200GB RAM. These datasets can take several hours, on the order of 2-3 days to terminate. If your computer does not have sufficient RAM memory or more applications have reserved memory, this might be killed by the OS. This is a well-known limitation of the compiler that consumes large amounts of memory.

Running individual files in `datasets/*` Our script will generate a C file from the dataset file including non-deterministic checks. We compile it with the Otti compiler, prove and verify it and print Verification successful and the total runtime. of each stage. Note that running indiviudal SGD datasets not from PLMB is not supported at this time.

A.5 Experiment workflow

Our experiment runs a script around the components of Otti to compile publicly available datasets to zkSNARKS and then verifies them, printing "Verification successful" upon completion. We also output profiling information such as runtime and zkSNARK proof size.

A.6 Evaluation and expected results

In Otti we evaluate the practicality of compiling numerical optimization problems to zkSNARKs. We evaluate Otti in linear programming, semi-definite programming, and stochastic optimization problems. We apply this technique to publicly available datasets [1, 3, 4], and show the following results.

A.6.1 Semidefinite programming results

Dataset	Prover (ms)	Verifier (ms)	Proof (KB)	Solver (ms)	R1CS constraints
truss1	5140	768	79.20	197	3,007,933
hinf1	7166	1209	79.88	215	4,703,942
hinf2	10607	1187	79.88	313	6,536,398
hinf3	7795	1038	79.88	362	6,536,398
hinf4	9008	1211	79.88	193	6,536,398
hinf5	7748	1248	79.88	238	6,536,398
hinf6	7051	912	79.88	294	6,536,398
hinf7	7432	1058	79.88	343	6,536,398
hinf8	7241	1105	79.88	321	6,536,398
hinf9	7546	1153	79.88	301	6,536,398
control1	7398	1069	79.88	181	6,968,254

A.6.2 Linear programming evaluation results

Dataset	Prover (ms)	Verifier (ms)	Proof (KB)	Solver (ms)	R1CS constraints
afiro	318	73	19.82	41	36,811
sc50a	320	78	19.82	42	54,066
sc50b	336	77	19.82	40	55,085
adlittle	609	117	29.33	45	180,747
sc105	473	104	20.51	45	113,282
scagr7	595	111	29.33	47	229,061
israel	1072	128	47.02	56	511,156
agg	2486	511	47.71	56	1,069,523
sc205	665	121	29.33	52	220,520
brandy	1631	227	47.02	61	815,356
beaconfd	2499	337	47.71	56	1,149,169
agg2	2237	313	47.71	79	1,887,762
agg3	2401	383	47.71	71	1,891,690
lotfi	1014	183	30.01	56	326,102
scorpion	1645	208	47.02	62	731,137
sctap1	1007	180	47.71	61	414,101
scfxm1	1831	254	47.02	105	965,504
bandm	2499	467	47.02	103	1,093,340
scagr25	1637	268	47.71	111	823,136
degen2	1534	223	47.71	308	626,407
scsd1	1636	216	47.02	54	1,034,359
fffff800	2431	330	47.71	197	1,479,725
scfxm2	2426	354	47.02	304	1,932,500
scrs8	2512	363	47.71	117	1,601,971
bnl1	4077	558	81.10	236	2,324,544
scsd6	2372	422	47.71	100	1,845,814
modszk1	2449	369	47.71	185	1,805,821
scsd8	4767	567	81.10	477	3,607,188

A.6.3 Stochastic gradient descent results

Dataset	Prover (ms)	Verifier (ms)	Proof (KB)	Solver (ms)	R1CS constraints
confidence	0.117	0.038	14.08	2.35	13,027
haberman	0.215	0.052	19.36	7.84	60,237
iris	0.293	0.076	11.47	4.13	4,730
new_thyroid	0.296	0.058	14.75	2.96	25,810
krkopt	0.997	0.125	29.31	39.70	399,555
diabetes	0.484	0.071	28.64	32.14	212,501
glass	0.104	0.027	11.47	3.14	7,571
labor	0.186	0.047	14.75	3.19	22,763
letter	1.01	0.164	29.31	27.97	374,655
lymphography	0.284	0.055	14.75	4.37	31,823
collins	0.323	0.08	14.75	4.23	31,733
allbp	0.301	0.055	20.03	11.35	103,451
dermatology	0.517	0.106	19.36	6.90	55,877
kddcup	0.904	0.147	28.64	67.80	198,840
molecular_biology_promoters	0.707	0.263	19.36	7.19	41,343
mfeat_karhunen	0.488	0.073	28.64	13.80	162,352
analcatdata_authorship	0.586	0.095	28.64	8.70	231,455
clean1	6.423	0.535	79.20	14.47	3,473,740
clean1 (50%)	4.675	0.607	79.20	14.47	2,262,837
clean2 (50%)	18.234	1.337	79.88	477.17	6,773,944
GE1000 (50%)	4.842	0.356	45.92	310.64	571,558

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

References

[1] LP/data index. https://ampl.com/netlib/lp/data/, 2013.

[2] lpsolve: Mixed integer linear programming (MILP) solver. http://lpsolve.sourceforge.net/5.5, 2021.

[3] B. Borchers. SDPLIB 1.2, a library of semidefinite programming test problems. *Optimization Methods and Software*, 11(1-4), 1999.

[4] J. D. Romano, T. T. Le, W. La Cava, J. T. Gregg, D. J. Goldberg, P. Chakraborty, N. L. Ray, D. Himmelstein, W. Fu, and J. H. Moore. PMLB v1.0: an open-source dataset collection for benchmarking machine learning methods. *Bioinformatics*, 38(3):878–880, 10 2021.



A Artifact Appendix

A.1 Abstract

This artifact reproduces distributed systems experiments that benchmark the *collaborative zkSNARKs* that we evaluate in our paper.

Some experiments run on Google Cloud Platform, so we will give the evaluators SSH access to one of our machines which has appropriate credentials to launch the experiments.

The artifact includes scripts to re-run a limited version of our experiments, and to re-render our plots.

A.2 Artifact check-list (meta-information)

- **Algorithm:** GSZ20 and SPDZ MCP protocols. Groth16, Marlin, and PLONK zkSNARKs.
- **Compilation:** Rust compiler, nightly.
- **Run-time environment:** Linux, some experiments on GCP
- **Execution:** A distributed protocol
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 30 minutes
- **Publicly available (explicitly provide evolving version reference)?:** Yes, at `https://github.com/alex-ozdemir/multiprover-snark`.
- **Archived (explicitly provide DOI or stable reference)?:** `https://github.com/alex-ozdemir/multiprover-snark/tree/98cc63c7b885ade04989a5505050504ae7f2aac0`

A.3 Description

A.3.1 How to access

The source is available at `https://github.com/alex-ozdemir/multiprover-snark/tree/98cc63c7b885ade04989a5505050504ae7f2aac0`.

We will provide access to a machine that can run the experiments.

A.3.2 Hardware dependencies

At least 8GB of RAM.

A.3.3 Software dependencies

Ubuntu packages: zsh libgmp-dev neovim autoconf pkg-config libtool apache2-dev apache2 dnsmasq-base protobuf-compiler libprotobuf-dev libssl-dev libxcb-present-dev libcairo2-dev libpango1.0-dev tmux units r-base

Rust compiler: nightly after 2022-01-31.

Ripgrep

Mahimahi network emulator, patched as described in the source distribution at `/mpc-snarks/artifact_eval.md`

R libraries: ggplot2, dplyr, readr, scales

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

(You can skip this. We'll give you access to a machine with the software set up, and alternatively to a VM that is already set up.)

1. New machine, at least 8GB RAM, 10GB disk
   - Ubuntu 20.04 server
2. Do Ubuntu installation
   - username: user password: user
   - updating took a while
3. `apt install zsh libgmp-dev neovim autoconf pkg-config libtool apache2-dev apache2 dnsmasq-base protobuf-compiler libprotobuf-dev libssl-dev libxcb-present-dev libcairo2-dev libpango1.0-dev tmux units r-base virutalbox-guest-utils`
4. `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`
   - nightly
5. `cargo install ripgrep`
6. Install the mahimahi shell network emulator
   - clone it
   - apply patches
     - empty PICKY_CXXFLAGS in configure.ac (compiler is pickier now)
     - add mm-rate-to-events to install list in scripts/Makefile.am (need this)
   - `./autogen.sh && ./configure && make -j 8`
   - `sudo sysctl -w net.ipv4.ip_forward=1`
7. Install R libraries: ggplot2, dplyr, readr, scales
8. Set up folder sharing `sudo adduser user vboxsf && sudo systemctl enable virutalbox-guest-utils.service`

A.5 Experiment workflow

1. Give us your public key using HotCRP.
2. Wait for us to confirm that we have granted that key access.
3. `ssh aeval@128.12.176.8`
4. `cd ~/multiprover-snark/mpc-snarks`
5. Run `git clean -fd` to clear any existing data.
6. Check that `git rev-parse HEAD` outputs 98cc63c7b885ade04989a5505050504ae7f2aac0.
7. `cargo build --release`

- You can `cargo clean` first to force a clean build.

5. Optional: run the test suite `./test.zsh`

- If it exits with a zero return code, it was successful.

Now, run the experiments (next section)

A.6 Evaluation and expected results

1. Run all experiments with `time ./analysis/collect/artifact_eval.zsh`

- This should take approximately 24 minutes.
- Alternatively: you can run the experiments one-by-one:

    1. `time ./analysis/collect/bad_net.zsh | tee ./analysis/data/bad_net.csv`

    – This runs locally and should take approximately 6 minutes

    2. `time ./analysis/collect/weak_machines.zsh`

    – This runs on GCP and should take approximately 10 minutes

    3. `time ./analysis/collect/Npc.zsh`

    – This runs on GCP should take approximately 8 minutes

2. Generate all plots: `./analysis/plotting/artifact_eval.zsh`
3. Copy plots to your machine: `scp 'aeval@128.12.176.8:multiprover-snark/mpc-snarks/analysis/plots/*.pdf' .`
4. Analyze:

    1. Varying constraint counts: mpc.pdf should be comparable to Figure 8
        - At large constraint counts, 3PC GSZ should have runtime similar to "Single Prover". The SPDZ MPCs should have approximately twice the runtime.
    2. Varying prover count: Npc.pdf should be comparable to Figure 9
        - Both SPDZ and GSZ should be parabolas. Slowdown should be ~2x and ~1x respectively for 2 parties.
    3. Varying link capacity: bad_net.pdf should be comparable to Figure 10
        - Slowdown should be going to ~2x as bandwidth increases. Plonk should be slower than the others.

A.7 Experiment customization

If you want to reproduce the single-machine experiments (those that vary link capacity using a network emulator) on your machine, follow the directions below.

This is optional. You have already produced this graph on our machine.

A.7.1 Build the collaborative proofs

Download the VM here: https://doi.org/10.5281/zenodo.5889564. User: user. Password: user.

1. `cd ~`
2. `git clone -b artifact-eval https://github.com/alex-ozdemir/multiprover-snark`
3. `cd multiprover-snark/mpc-snarks`
4. `cargo build --release`
5. Optional: run the test suite `./test.zsh`

- If it exits with a zero return code, it was successful.

A.7.2 Collect the data

1. `time ./analysis/collect/bad_net.zsh | tee ./analysis/data/bad_net.csv`

- This should take approximately 6 minutes

A.7.3 Make & inspect the plots

1. Varying numbers of provers

- Run: `Rscript ./analysis/plotting/bad_net.R`
- Output plot: ./analysis/plots/bad_net.pdf
- It should be comparable to Figure 10

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

This artifact is to help users reproduce the results we reported in our *USENIX Security 2022* paper submission. We recommend to run the artifact on a **x86-64** computer with ≥ 20 CPU cores, $\geq 600GB$ of memory and $\geq 1.5TB$ hard drive storage, and with an **Ubuntu 20.04 LTS** operating system. The artifact should reproduce all the Figures and Tables we reported in the paper, and thus can validate the main claims of the paper. Detailed execution steps are elaborated in the artifact *README.md* file.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Coverage-based fuzzing, validity-oriented query mutation and DBMS oracle.
- **Program:** `SQLRight`. The program source code is included in the artifact.
- **Compilation:** afl-clang-fast and gcc-9/g++-9.
- **Binary:** Binaries not included. The programs are built from source.
- **Run-time environment:** OS: Ubuntu 20.04 LTS. Dependencies: python3 runtime and Docker. Requires Root access.
- **Hardware:** A **x86-64** computer with ≥ 20 CPU cores, $\geq 600GB$ of memory and $\geq 1.5TB$ hard drive storage. Hardware specs are publicly available.
- **Metrics:** The reported metrics are: Number of Bugs Detected, Fuzzing Coverage Feedback, Generated Query Validity and Number of Valid Statements per Hour.
- **Output:** All the Figures and Tables in the paper.
- **How much disk space required (approximately)?:** Around $1.0TB$ (10^{12} bytes).
- **How much time is needed to complete experiments (approximately)?:** Around 8834 CPU hours.
- **Publicly available (explicitly provide evolving version reference)?:** Publicly available on Github.
- **Code licenses (if publicly available)?:** MIT License
- **Archived (explicitly provide DOI or stable reference)?:** Yes. Stable reference: https://github.com/psu-security-universe/sqlright-artifact/tree/57978e5ce697e13414a2bca871d2ef874e77158d

A.3 Description

A.3.1 How to access

The artifact can be retrieved from Github.

The Github link to the artifact is: https://github.com/psu-security-universe/sqlright-artifact/tree/57978e5ce697e13414a2bca871d2ef874e77158d.

A.3.2 Hardware dependencies

The artifact evaluations are run on a **x86-64** computer, recommended with ≥ 20 CPU cores, $\geq 600GB$ of memory and $\geq 1.5TB$ hard drive storage.

A.3.3 Software dependencies

The artifact is evaluated on an **Ubuntu 20.04 LTS** operating system.

A.3.4 Data sets

N/A.

A.3.5 Models

N/A.

A.3.6 Security, privacy, and ethical concerns

N/A.

A.4 Installation

To run the artifact code, user should download the artifact files from the Github website (link provided from above). The *README.md* file contains the detailed instructions to install the Docker environment, and further build the Docker Images required for the fuzzing tests.

A.5 Experiment workflow

The experiments are being hosted inside the Docker virtualized environment. User only needs to call a few scripts guided by the *README.md* file, and the scripts will run the fuzzing evaluations in the background and later generate all the Figures and the Tables we presented in the paper.

A.6 Evaluation and expected results

Here is the main claims of the paper:

- The proposed tool `SQLRight` can find more bugs than State-of-the-arts `SQLancer` and $\texttt{Squirrel}_{+\texttt{oracle}}$. `SQLRight` also outperforms existing tools in triggering more program code. This claim can be validated by *Figure 5* and *Figure 8*.
- The Coverage-based guidance helps `SQLRight` generate more diverse queries and accumulate useful mutations, which helps discover more bugs than the no-feedback baselines. This claim can be validated by *Figure 6* and *Table 3*.
- The validity-oriented optimizations in SQLRight can help generate higher validity queries, reduce false positives, and ultimately help discover more bugs. This claim can be validated by *Figure 7*, *Figure 9* and *Table 4*.

Following the instructions provided by the *README.md* files in the artifact, one should be able to independently reproduce all the results (Figures, Tables) shown in our paper. Specifically:

- **Session 3** in the *README.md* contains the instructions to evaluate Comparison with Existing Tools (*Section 5.2* in the paper). It includes the steps to generate the figures from *Figure 5* and *Figure 8* in the paper. It consumes about 6152 CPU hours.
- **Session 4** in the *README.md* contains the instructions to evaluate Contribution of Coverage Feedback (*Section 5.3* in the paper). It includes the steps to generate *Figure 6* and *Table 3* in the paper. It consumes about 726 CPU hours.
- **Session 5** in the *README.md* contains the instructions to evaluate Contribution of Validity (*Section 5.4* in the paper). It includes the steps to generate *Figure 7*, *Figure 9* and *Table 4* in the paper. It consumes about 1956 CPU hours.

The detailed command instructions are elaborated in the *README.md* file. Here we show the expectations for each artifact generated figures/tables:

- **Figure 5a** `SQLite` logical bugs: `SQLRight` should detect the most bugs. On different evaluation around, we expect ≥ 3 bugs being detected by SQLRight in 72 hours.
- **Figure 5b** `MySQL` logical bugs: The current bisecting and bug filtering scripts could slightly over-estimate (or under-estimate) the number of unique bugs for MySQL. Some manual efforts might be needed to scan through the bug reports and deduplicate the bugs to get the most accurate unique bug number. But in general, `SQLRight` should report the most bugs after bisecting (≥ 2 bugs in 72 hours).
- **Figure 5c-e** `SQLite`, `MySQL` and `PostgreSQL` code coverage: `SQLRight` should have the highest code coverage among the other baselines.
- **Figure 5f** `SQLite` query validity: `SQLancer` has the highest query validity, while `SQLRight` performs better than $\texttt{Squirrel}_{\texttt{+oracle}}$.
- **Figure 5g** `MySQL` query validity: sys has higher validity than $\texttt{Squirrel}_{\texttt{+oracle}}$.
- **Figure 5h** `PostgreSQL` query validity: `SQLancer` has the highest query validity, while `SQLRight` performs better than $\texttt{Squirrel}_{\texttt{+oracle}}$.
- **Figure 5i** `SQLRight` valid statements per hour: `SQLancer` has the highest number of valid statements per hour, while `SQLRight` performs better than $\texttt{Squirrel}_{\texttt{+oracle}}$.
- **Figure 5j** `MySQL` valid statements per hour: `SQLRight` has more valid statements per hour than $\texttt{Squirrel}_{\texttt{+oracle}}$.
- **Figure 5k** `MySQL` valid statements per hour: `SQLancer` have the highest valid statements per hour, while `SQLRight` performs better than $\texttt{Squirrel}_{\texttt{+oracle}}$.
- **Figure 6a-b** bugs of `SQLite` (`NoREC` and `TLP`): `SQLRight` should detect the most bugs. On different evaluation around, we expect ≥ 2 bugs being detected by `SQLRight` in 24 hours.
- **Figure 6c-d** coverage of `SQLite` (`NoREC` and `TLP`): `SQLRight` should have the highest code coverage among the other baselines.
- **Figure 7a** `SQLite` logical bugs: `SQLRight` should detect the most bugs. On different evaluation around, we expect ≥ 2 bugs being detected by SQLRight in 24 hours. Additionally, we have muted the $\texttt{SQLRight}_{\texttt{-deter}}$ config in the Artifact logical bugs figure. Because sometimes $\texttt{SQLRight}_{\texttt{-deter}}$ could produce tens of False Positives, which would destroy the plot region and render the script outputs an unreadable plots.
- **Figure 7b** `MySQL` logical bugs: The current bisecting and bug filtering scripts could slightly over or under-estimate the number of unique bugs for MySQL. Some manual efforts might be needed to scan through the bug reports and deduplicate the bugs to get the most accurate unique bug number. In general, SQLRight should report the most bugs after bisecting. On different evaluation around, we expect ≥ 1 bugs from `SQLRight` in 24 hours. Additionally, we have muted the $\texttt{SQLRight}_{\texttt{-deter}}$ config in the Artifact logical bugs figure. Because sometimes $\texttt{SQLRight}_{\texttt{-deter}}$ could produce tens of False Positives, which would destroy the plot region and render the script outputs an unreadable plots.
- **Figure 7c-e** `SQLite` code coverage: `SQLRight` and $\texttt{SQLRight}_{\texttt{-deter}}$ should have the highest code coverage among the other baselines. $\texttt{SQLRight}_{\texttt{-ctx-valid}}$ could have a coverage very close to the SQLRight config, but in general, $\texttt{SQLRight}_{\texttt{-ctx-valid}}$ is slightly worse in coverage compared to `SQLRight`.
- **Figure 7f-h** `SQLRight` and $\texttt{SQLRight}_{\texttt{-deter}}$ should have the highest query validity.
- **Figure 7i-k** `SQLRight` and $\texttt{SQLRight}_{\texttt{-deter}}$ should have the highest number of valid statements per hour.
- **Figure 8a** `SQLite` logical bugs: `SQLRight` should detect the most bugs. On different evaluation around, we expect ≥ 1 bugs being detected by SQLRight in 72 hours.
- **Figure 8b** `MySQL` logical bugs: The current bisecting and bug filtering scripts could slightly over-estimate (or under-estimate) the number of unique bugs for MySQL. Some manual efforts might be needed to scan through the bug reports and deduplicate the bugs to get the most accurate unique bug number. But in general, `SQLRight` should reported the most bugs after bisecting (≥ 1 bugs in 72 hours).
- **Figure 8c-8e** `SQLite`, `MySQL` and `PostgreSQL` code coverage: `SQLRight` should have the highest code coverage among the other baselines.
- **Figure 8f-h** `SQLite`, `MySQL` and `PostgreSQL` query validity: `SQLancer` has the highest query validity, while `SQLRight` performs better than $\texttt{Squirrel}_{\texttt{+oracle}}$.
- **Figure 8i-k** `SQLite`, `MySQL` and `PostgreSQL` valid statements per hour: `SQLancer` has the highest number of valid statements per hour, while `SQLRight` performs better than $\texttt{Squirrel}_{\texttt{+oracle}}$.
- **Figure 9a** `SQLite` logical bugs: `SQLRight` should detect the most bugs. On different evaluation around, we expect ≥ 2 bugs being detected by `SQLRight` in 24 hours. Additionally, we have muted the $\texttt{SQLRight}_{\texttt{-deter}}$ config in the Artifact logical bugs

figure. Because sometimes SQLRight$_{-deter}$ could produce tens of False Positives, which would destroy the plot region and render the script outputs an unreadable plots.

- **Figure 9b** MySQL logical bugs: The current bisecting and bug filtering scripts could slightly over or under-estimate the number of unique bugs for MySQL. Some manual efforts might be needed to scan through the bug reports and deduplicate the bugs to get the most accurate unique bug number. In general, SQLRight should detect the most bugs after bisecting. On different evaluation around, we expect ≥ 1 bugs being reported by SQLRight in 24 hours. Additionally, we have muted the SQLRight$_{-deter}$ config in the Artifact logical bugs figure. Because sometimes SQLRight$_{-deter}$ could produce tens of False Positives, which would destroy the plot region and render the script outputs an unreadable plots.
- **Figure 9c-e** SQLite, MySQL and PostgreSQL code coverage: SQLRight and SQLRight$_{-deter}$ should have the highest code coverage among the other baselines. SQLRight$_{-ctx-valid}$ could have a coverage very close to SQLRight, but in general, SQLRight$_{-ctx-valid}$ is slightly worse in coverage compared to SQLRight.
- **Figure 9f-h** SQLite, MySQL and PostgreSQL query validity: SQLRight and SQLRight$_{-deter}$ should have the highest query validity.
- **Figure 9i-h** SQLite, MySQL and PostgreSQL valid statements per hour: SQLRight and SQLRight$_{-deter}$ should have the highest number of valid statements per hour.
- **Table 3** Code coverage triggered by queries with different depths: The mutation depth number could be slightly different between each run. However, the **Max Depth** from SQLRight NoREC and TLP should be larger than other baselines. And SQLRight NoREC and TLP should have more queue seeds located in a deeper depth, compared to other baselines.
- **Table 4** False Positives from **Non-Deter**: We have introduced some extra filters that can filter out some obvious False Positives. We includes these filters in the Artifact implementation, in order to reduce the manual efforts for excluding FPs, and to produce a more accurate bug numbers by default. Therefore, the bug number reported by the current Artifact script could be slightly different from the ones we reported in the paper (**Table 4**). For all configurations, the **WITHOUT non-deter** settings should always have less bugs reported compared to the **WITH non-deter** settings, due to the extra False Positives produced by the non-deterministic queries.

A.7 Experiment customization

N/A

A.8 Notes

N/A

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.



A Artifact Appendix

A.1 Abstract

This artifact provides the source code of ASan--, a tool assembling a group of optimizations to reduce (or "debloat") sanitizer checks and improve ASan's efficiency. It also provides a set of test cases and necessary dependencies. In principle, ASan-- has no special requirment of the hardware or operating system. For reproductive experiments, we recommend reviewers to build ASan-- on Ubuntu 18.04 LTS 64bit (a virtual machine is fine) and we suggest reviewers to install the desktop version of Ubuntu for Chromium evaluation. For software requirements, we separate into two parts. For runtime evaluation, we used SPEC CPU2006 and Chromium to test the performance of our tool. For bug detection capability evaluation, we utilized Juliet Test Suite and Linux Flaw Project to detect the vulnerabilities in software, and ASan-- should achieve identical results with ASan.

A.2 Artifact check-list (meta-information)

- **Algorithm:**
  Yes, we present six new algorithms. Details can be found in our paper.
- **Program:**
  SPEC CPU2006: is private, and it should be downloaded. The approximate size is 20GB.
  Chromium: is public, and version 58.0.3003.0 should be downloaded. The approximate size is 50GB.
  Juliet Test Suite: is public, and it is included in our artifact. The approximate size is 500MB.
  Linux Flaw Project: is public, and it is included in our artifact. The approximate size is 1.5GB.
- **Compilation:**
  The compiler we used is LLVM-4.0.0, which is public, and it is included in our artifact.
- **Run-time environment:**
  We recommend Ubuntu 18.04 LTS 64bit for testing, and you will need root access.
- **Execution:**
  The experiments will approximately run 5 hours.
- **Metrics:**
  Execution time and Vulnerabilities reproduction.
- **Output:**
  The outputs will be numerical results and error logs.
- **Experiments:**
  We provided both bash scripts and manual steps for users to reproduce results.
- **How much disk space required (approximately):**
  Approximately 100GB.
- **How much time is needed to prepare workflow (approximately):**
  The estimate time to prepare workflow will be 8 hours.
- **How much time is needed to complete experiments (approximately):**
  The estimate time to complete experiments will be 15 hours.
- **Publicly available:**
  Yes, our artifact is publicly available.
- **Archived (provide DOI or stable reference):**
  Yes, our artifact is publicly available on GitHub.

A.3 Description

A.3.1 How to access

Clone below repository from GitHub
https://github.com/junxzm1990/ASAN--/tree/f497310328fafddc7fe7993edb8befd4ab4d6393

A.3.2 Hardware dependencies

The approximate disk space required after unpacking our artifact is 100GB.

A.3.3 Software dependencies

For the OS, we recommend Ubuntu 18.04 LTS 64bit and we suggest reviewers to install the desktop version of Ubuntu. For software dependencies, we separate into two parts. For run-time evaluation, we used SPEC CPU2006 and Chromium. For bug detection capability evaluation, we utilized Juliet Test Suite and Linux Flaw Project.

A.4 Installation

You can get the artifacts from GitHub using the following command:

```
git clone https://github.com/junxzm1990/ASan--.git
```

A.5 Experiment workflow

The overall workflow consists of the following steps:

1. Install the dependencies;
2. Build tool ASan--;
3. Build and run the SPEC CPU2006 benchmarks;
4. Build and run the Chromium benchmarks;
5. Build and run the Juliet Test Suite benchmarks;
6. Build and run the Linux Flaw Project benchmarks.

We have provided scripts for each of the steps above.

A.6 Evaluation and expected results

We will go through the entire experiment workflow by describing all the commands in each step.

A.6.1 Install the dependencies

Run the following commands:

```
sudo apt-get install cmake
sudo apt-get install git
sudo apt-get install wget
sudo apt-get install tar
```

A.6.2 Build tool ASan--

Run the following commands:

```
git clone https://github.com/junxzm1990/ASan--.git
cd ASan--
cd llvm-4.0.0-project
mkdir ASan--Build && cd ASan--Build
cmake -DLLVM_ENABLE_PROJECTS="clang;compiler-rt" -G
"Unix Makefiles" ../llvm
make -j
```

A.6.3 Build fuzzing version ASan--

Run the following commands:

```
patch -p1 < patch_ASanASan--
cd llvm-4.0.0-project
mkdir ASanASan--Build && cd ASanASan--Build
cmake -DLLVM_ENABLE_PROJECTS="clang;compiler-rt" -G
"Unix Makefiles" ../llvm
make -j
```

A.6.4 Build and run the AFL Fuzzing

Run the following commands:

```
cd /fuzzing
bash set_ASan--.sh
cd binutils-2.32
bash auto_build_ASan--.sh
./afl-2.52b/afl-fuzz -S nm_afl -i
./afl-2.52b/testcases/others/elf/ -o ./eval/nm -m
none - ./binutils-2.32/ASan_Srk/binutils/nm-new @@
```

The result will be printed on the console.

A.6.5 Build and run the SPEC CPU2006

Run the following commands:

```
cd /spec
cp run_asan--.sh /cpu2006
cd cpu2006
CC=PATH/llvm-4.0.0-project/ASan--Build/bin/clang
CXX=PATH/llvm-4.0.0-project/ASan--Build/bin/clang++
./run_asan--.sh asan-- <test|train|ref> <int|fp>
```

The result will be printed on the console.

A.6.6 Build and run the Chromium

Run the following commands:

```
cd /chromium
git clone https://chromium.googlesource.com
/chromium/tools/depot_tools.git
```

Add depot_tools to the end of your PATH :

```
export PATH="$PATH:/path/to/depot_tools"
```

Create a chromium directory for the checkout and change to it

```
mkdir  /chromium && cd  /chromium
```

Run the fetch tool from depot_tools to check out the code and its dependencies.

```
fetch -nohooks chromium
git checkout tags/58.0.3003.0 -b 58
```

Check out a version of depot_tools from around the same time as the target revision.

```
# Get date of current revision:
/chromium/src $ COMMIT_DATE=$(git log -n 1
-pretty=format:%ci)
# Check out depot_tools revision from the same time:
/depot_tools $ git checkout $(git rev-list -n 1
-before="$COMMIT_DATE" <main | master>
/depot_tools $ export DEPOT_TOOLS_UPDATE=0
```

Checkout all the submodules at their branch DEPS revisions.

```
gclient sync -D -force -reset -with_branch_heads
```

To create a build directory, run:

```
gn args out/ASan--
```

Set build arguments.

```
is_clang = true
clang_base_path = "llvm-4.0.0-project/ASan--Build"
is_asan = true
is_debug = ture
symbol_level = 1
is_component_build = true
pdf_use_skia = true
```

Build Chromium (the “chrome” target) with Ninja:

```
ninja -C out/ASan-- chrome
```

Run benchmarks and Reproduce bugs:

```
# Sunspider:
./chrome https://webkit.org/perf/sunspider-0.9.1/
sunspider-0.9.1/driver.html
# Kraken:
./chrome https://mozilla.github.io/krakenbenchmark.
mozilla.org/index.html
# Lite Brite:
./chrome https://testdrive-archive.azurewebsites.net/
Performance/LiteBrite/
# Octane:
./chrome https://chromium.github.io/octane/
# Basemark:
./chrome https://web.basemark.com/
# WebXPRT:
./chrome https://www.principledtechnologies.com/
benchmarkxprt/webxprt/run-webxprt-mobile
```

#Issue 848914:

```
./chrome -disable-gpu /Issue_848914_PoC/gpu_freeids.html
```

#Issue 1116869:
```
./chrome /Issue_1116869_PoC/poc_heap_buffer_overflow
```
#Issue 1099446:
```
./chrome /Issue_1099446_PoC/poc_heap_buffer_overflow
```
The result will be printed on the console.

A.6.7 Build and run the Juliet Test Suite

Juliet contains different benchmarks, here we take CWE121 as an example. Run the following commands:
```
cd /juliet_test_suite
cd testcases
cd CWE121_Stack_Based_Buffer_Overflow
cd s01
make -j
export ASAN_OPTIONS=halt_on_error=0
./CWE121_s01
```
The result will be printed on the console.

A.6.8 Build and run the Linux Flaw Project

Linux Flaw contains different benchmarks, here we take CVE-2006-0539 as an example. Run the following commands:
```
cd /linux_flaw_project
```

Install the dependencies.
```
sudo apt-get install sendmail
sudo apt-get install vim
sudo apt-get install pkg-config
sudo apt-get install fontconfig
sudo apt-get install libfontconfig1-dev
export CC=$(readlink -f ../../llvm-4.0.0-project
/ASan--Build/bin/clang)
CXX=$(readlink -f ../../llvm-4.0.0-project
/ASan--Build/bin/clang++)
```

Build and run.
```
wget https://github.com/mudongliang/source-packages/
raw/master/CVE-2006-0539/fcron-3.0.0.src.tar.gz
tar -xvf fcron-3.0.0.src.tar.gz
cp configure ./fcron-3.0.0
cd fcron-3.0.0
sudo mkdir /var/spool/fcron
CC=$CC CXX=$CXX CFLAGS="-fsanitize=address -g"
CXXFLAGS="-fsanitize=address -g" ./configure
make -j
./convert-fcrontab `perl -e 'print "pi3"x600'`
```
The result will be printed on the console.



A Artifact Appendix

A.1 Abstract

We provide code, data, and outputs of our experiments. Our artifact is publicly available at https://github.com/Yuanyuan-Yuan/Manifold-SCA with detailed documents. Using our tool, users can perform side channel attacks on media software and localize side channel vulnerabilities of the target software. We also provide a mitigation scheme towards our attack and investigate the noise resilience of our attacking technique.

A.2 Artifact check-list (meta-information)

- **Data set.** See README in our artifact.
- **Run-time environment.** Our experiments are launched on 64-bit Ubuntu 18.04, we recommend users to set up on the same OS. We also provide a docker container with everything set up. Performing `Prime+Probe` attack needs root access.
- **Hardware.** We perform `Prime+Probe` attacks on Intel Xeon and AMD Ryzen CPUs. Nevertheless, our approach is *not* hardware-specific. Users can use our tools on other CPUs. To approximate manifold from known data (i.e., the training split), users are recommended to run scripts on GPUs. Note that our tool requires a relatively large RAM.
- **Execution.** Our experiments are launched on one Nvidia GeForce RTX 2080 GPU. The running time of approximating manifold (i.e., training models) is less than 24 hours. Nevertheless, it will be very slow if the script is executed with only CPUs. We have released our trained models. The data processing and side channel logging are also time-consuming, which may take several days. We also provide our processed data and logged side channels.
- **Security, privacy, and ethical concerns.** Our tool is provided as-is and is only for research purposes. Please use it only on test systems with no sensitive data. Users are responsible for protecting themselves, their data, and others from potential risks caused by our tool.
- **Output.** Our outputs include 1) logged side channel records; 2) trained models which appropriate data manifold; 3) reconstructed media data from unknown side channel; 4) localized side channel vulnerabilities of media software.

  We release 1) scripts for logging side channels and our logged side channel records; 2) scripts for training models and our trained models; 3) scripts for reconstructing media data from unknown side channels (i.e., the test split) and our reconstructed media data; 4) scripts for localizing side channel vulnerabilities and our localized vulnerabilities. Some vulnerabilities have been explored by previous works, and the new-found vulnerabilities have been confirmed by developers of `FFmpeg` and `libjpeg` by the time of writing. See outputs for more details.
- **How much disk space required (approximately)?** We provide 1K samples of processed data and side channel records for each dataset and software. We also provide our trained models and a docker container. To launch experiments using these data samples, which are sufficient to verify our findings, users need to prepare **at least 20G** space. Further, if users want to prepare all data (we also provide the scripts), **2T** space is desired.
- **Experiments.** We provide 1K samples of processed data and side channel records for each dataset and software. These samples are sufficient to verify our statements and results, for instance, reconstructing high-quality media data from side channel records and mitigating side channel attack using perception blinding (e.g., perceptual properties of reconstructed images are dominated by the mask).

  Since experiments are performed on a limited number of data, some numerical results may have relatively large variances. Also, it's worth noting that the 1K samples are *not* enough to train the model (i.e., the trained model has a poor capability of reconstructing media from unknown side channels), but users can see that the reconstructed media data from known side channels gradually have higher quality and get similar to the reference media data. Users can use our provided scripts to produce all data involved in our paper.
- **How much time is needed to prepare workflow/complete experiments (approximately)?**

  1) Set up the environment: less than 1 hour. We also provide a docker container with everything set up.

  2) Download public datasets and process: it requires less than 1 hour to process the data. We provide our processed data samples.

  3) Log side channels: around one week to log side channels of all media and target software. We provide our logged side channels.

  4) Train models: training one model requires less than 24 hours on one Nvidia GeForce RTX 2080 GPU. Our script also supports training on CPUs, but it could be time-consuming. We also release our trained models.

  5) Others: a few minutes.
- **Publicly available?** Our artifact is publicly available at https://github.com/Yuanyuan-Yuan/Manifold-SCA.
- **Code licenses (if publicly available)?** MIT license.
- **Data licenses (if publicly available)?** CC-BY-4.0 license.
- **Workflow frameworks used?** We use Pytorch as the building block of our framework.
- **Archived (provide DOI or stable reference)?** Available at https://zenodo.org/record/5816702#.YdQMHxNByjA.

A.3 Description

See all details in our README.

A.3.1 How to access

Access our artifact at https://github.com/Yuanyuan-Yuan/Manifold-SCA.

A.4 Installation

See README. We also provide a docker container with everything set up.

A.5 Evaluation and expected results

We show that side channel analysis (SCA) towards media software can be largely boosted by manifold learning, which recasts SCA as mapping between side channels and media data via a low-dimensional joint manifold. Enabled by the neural attention mechanism, we can localize side channel vulnerabilities of media software by investigating which records on a logged side channel trace contribute most to the reconstruction of media data. Our findings have been confirmed by the software developers. We further propose the perception blinding that is highly effective for mitigating manifold learning-based side channel attacks. We also show that our approach is highly robust to noise in collected side channels.

Side Channel Attack. By using our released tools, users can log side channel records of the target software when it is processing private data. Based on the collected side channels and corresponding media data, users can train a model to appropriate the manifold. The trained model can reconstruct high-quality media data from unknown side channels (i.e., the test split of each dataset). We provide our trained models and 1K data samples (from test split). Using our trained models, users can observe that the reconstructed media data manifest consistent perceptual properties with the reference media data. Note that some numerical results (e.g., the text inference accuracy) may have large variances since they are calculated on only a few samples. Also, the provided data samples are *not* enough for training models, but users can still observe that the manifold (despite its poor generalization capability) is gradually formed when training models on these samples. To prepare all data records, which require roughly 2T space, users can download the public datasets and process them using our scripts. It's worth noting that due to the non-deterministic operations of Pytorch, training results and some inference results may be slightly different each time, but findings and conclusions derived from these results are consistent. Moreover, the results always largely outperform the baseline.

Localizing Side Channel Vulnerabilities. Users can use our scripts to localize side channel vulnerabilities once the manifold is formed. For instance, to investigate records produced by which functions in `libjpeg` contribute most to reconstructing images, users are expected to observe that `idct` and `mcu` related functions have the highest frequency. We also provide our localized vulnerabilities. Some vulnerabilities have been exported by previous works, and the new-found ones have been confirmed by developers. Note that the produced results on the 1K samples may be slightly different from our provided results. That is reasonable since the frequency of each localized function could have a relatively large variance on only a few examples.

Perception Blinding. We provide scripts for users to perform perception blinding on media data. We also provide blinded images and corresponding side channel records. Users can observe that given the side channels of blinded images, our framework can hardly reconstruct privacy—the perceptual properties are dominated by the blinding masks. Users can also use our scripts to produce other blinded data and use their customized blinding masks.

Noise Resilience. We show that our technique is robust towards noise in collected side channels. By using our provided scripts, users can introduce noise of various types and weights into side channels. The reconstructed media data from noisy side channels are still of high quality and manifest most of the perceptual properties of reference media data.

A.6 Experiment customization

Our artifact supports customized settings. More specifically, users can customize 1) the media datasets, 2) hardware platforms, 3) target software, 4) model architectures, 5) training parameters when appropriating manifold, 6) blinding masks, 7) noise insertion schemes. We provide APIs for customized settings; see details in README.





A Artifact Appendix

A.1 Abstract

Obligatory. Briefly describe your artifact including minimal hardware and software requirements, how it supports your paper, how it can be validated, and what is the expected result. At submission time, it will also be used to select appropriate reviewers. It will also help readers understand what was evaluated and how.

This artifact includes the source code for the experiments in the paper. The artifact is built upon Python and its libraries (e.g., Pytorch) and requires the access to GPUs for accelerating the model training. The required Python libraries are listed in the source code. The artifact is tested on Linux with NVIDIA V100 GPUs. The artifact will validate the attack performance observed in the paper. By running the code, the artifact will output original models, pruned models, and print out the results (i.e., attack accuracy) of membership inference attacks and defenses on the models.

A.2 Artifact check-list (meta-information)

Obligatory. Fill in whatever is applicable with some keywords and remove unrelated items.

- **Algorithm:** The proposed MIA attack and defense is proposed and included in the source code.
- **Model:** The ResNet18, DenseNet121, VGG16, FC models are included.
- **Data set:** The access to the CIFAR10, CIFAR100, CHMNIST, SVHN, Location, Texas, Purchase datasets is included.
- **Hardware:** GPU is required to accelerate model training.
- **Metrics:** The prediction accuracy and attack accuracy are reported.
- **Output:** The model prediction accuracy and attack accuracy will be output.
- **Experiments:** The guide to reproduce the experiments is provided in README file.
- **How much disk space required (approximately)?:** For each dataset and neural network architecture, we need around 10GB-100GB disk space to store original models, pruned models, pruned models with defense, and the corresponding shadow models. To run all the experiments, around 2TB disk space is required to store all the models. To reduce the disk space requirement, we can delete the models that have been evaluated, since the models trained on different datasets and neural network architectures are independent.
- **How much time is needed to prepare workflow (approximately)?:** Less than 1 hour is needed to install all the Python libraries.
- **How much time is needed to complete experiments (approximately)?:** It takes around 2-3 hours to evaluate the attacks and defenses on a single experimental setting using an NVIDIA V100 GPU. The entire experiment settings include 7 datasets, 4 neural network architectures, 4 pruning approaches, and 5 sparsity levels, in total 255 pruned models.
- **Publicly available (explicitly provide evolving version reference)?:** The code is available at `github.com/Machine-Learning-Security-Lab/mia_prune`.
- **Code licenses (if publicly available)?:** The code is under MIT License.
- **Data licenses (if publicly available)?:** All datasets are publicly available.

A.3 Description

Obligatory. For inapplicable subsections (e.g., the "How to access" subsection when not applying for the "Artifacts Available" badge), please specify 'N/A'.

A.3.1 How to access

Clone repository from Github. Final stale URL: `github.com/Machine-Learning-Security-Lab/mia_prune/tree/v1.0.0`.

A.3.2 Hardware dependencies

GPU is required to accelerate the neural network training and membership inference attacks.

A.3.3 Software dependencies

Python 3 is required. The code is tested using Python 3.8. The required Python libraries (e.g., Pytorch) is provided in the requirement.txt file.

A.3.4 Data sets

All the datasets are publicly available. The repository contains all the link to the datasets.

A.3.5 Models

The code is provided to generate machine learning models.

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

Obligatory. Describe the setup procedures for your artifact targeting novice users (even if you use a VM image or access to a remote machine).

First, install Python 3.8 with a virtual environment. Second, install the required Python libraries in the requirement.txt file. Third, create a folder to store the downloaded datasets. Fourth, create a folder to store the trained and pruned models.

A.5 Experiment workflow

Describe the high-level view of your experimental workflow and how it is implemented, invoked and customized (if needed), i.e. some OS scripts, IPython/Jupyter notebook, portable CK workflow, etc. This subsection is optional as long as the experiment workflow can be easily embedded in the next subsection.

The workflow for MIA attacks is summarized as follow: 1) Train an original neural network. 2) Prune the model and fine-tune the model. 3) Conduct membership inference attacks on the pruned model. 4) Conduct membership inference attacks on the original model.

The workflow for MIA defenses is summarized as follow: 1) Train an original neural network. 2) Based on an original model, prune the model and fine-tune the model with defense. 3) Evaluate the performance of defense by conduct membership inference attacks on the pruned model with defense.

A.6 Evaluation and expected results

Obligatory. Start by listing the main claims in your paper. Next, list your key results and detail how they each support the main claims. Finally, detail all the steps to reproduce each of the key results in your paper by running the artifacts. Describe the expected results and the maximum variation of empirical results (particularly important for performance numbers).

The paper presents the following main claims. 1) Neural network pruning increases the privacy risks of pruned models in terms of membership inference attacks. 2) The proposed SAMIA has advantages in identifying the pruned models' prediction divergence by using finergrained prediction metrics. 3) The proposed PPB protects the fine-tuning process of neural network pruning by reducing the prediction gaps based on their KL-divergence distances.

The key results include: 1) membership inference attack accuracy of the pruned models is usually higher than that of the original models. 2) the proposed SAMIA attack achieves the highest attack accuracy in most cases compared with baseline attacks. 3) the proposed PPB defense is effective in protecting all pruning approaches from attacks and can reduce the attack accuracy.

The steps to reproduce the first key results include: 1) Train an original neural network. 2) Prune the model and fine-tune the model. 3) Conduct SAMIA attacks on the pruned model. 4) Conduct SAMIA attacks on the original model.

The steps to reproduce the second results include: 1) Derive the pruned models in the first key result. 2) Conduct SAMIA attacks and baseline attacks on the pruned models.

The steps to reproduce the third results include: 1) Derive the original models in the first key result. 2) Prune the model and fine-tune the model with PPB defense. 3) Conduct SAMIA attacks on the pruned models.

Detailed examples for running these experiments are provided in the README file.

A.7 Experiment customization

The dataset can be changed by modifying the dataset.py file. The neural network architecture can be changed by modifying the models.py file. The pruning method can be changed by modifying the pruner.py file.

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.





A Artifact Appendix

A.1 Abstract

Our artifact primarily implements and evaluates our proposed model inversion attack strategies– LOMIA and CSMIA. It includes our codebase for the above two attack strategies, datasets, and APIs to query target and attack models. While our codebase is heavily Python-dependent, it can run without any specific hardware requirements. In our *requirements* file, we list packages used in our codebase, and in the installation guideline, we describe details of the installation guideline and also include a *readme* file, where we add details step by step procedure to run and evaluate our artifact, i.e., each of the claims we make. We provide the APIs of the target as well as attack ML models, with instructions to perform each attack leveraging these APIs and training datasets. We expect to reproduce the results shown in the paper, although the attack models trained on different accounts with their optimization technique might cause slightly different results.

A.2 Artifact check-list (meta-information)

- **Algorithm: N/A**
- **Program: N/A**
- **Compilation: N/A**
- **Transformations: N/A**
- **Binary: N/A**
- **Model: Decision Tree (DT), Deepnet (DNN)**
- **Data set: Adult, GSS, FiveThirtyEight**
- **Run-time environment: Python (3.7.11)**
- **Hardware: N/A**
- **Run-time state: N/A**
- **Execution: N/A**
- **Security, privacy, and ethical concerns: N/A**
- **Metrics: Precision, Recall, Accuracy, F1 score, G-mean, MCC, FPR**
- **Output:**
- **Experiments: LOMIA, CSMIA (we report median of 5 runs, expected variation 2-3% for GSS, and 5-7% for FiveThirtyEight datasets)**
- **How much disk space required (approximately)?: N/A**
- **How much time is needed to prepare workflow (approximately)?: N/A**
- **How much time is needed to complete experiments (approximately)?: N/A**
- **Publicly available (explicitly provide evolving version reference)?: N/A**
- **Code licenses (if publicly available)?: N/A**
- **Data licenses (if publicly available)?: N/A**
- **Workflow frameworks used?: N/A**
- **Archived (explicitly provide DOI or stable reference)?: N/A**

A.3 Description

A.3.1 How to access

This artifact codebase is shared via Github. The codebase can be downloaded from there. Also, we share the target model training datasets for our experiments, and associated attack datasets as well as attack model APIs. Our target models (DT, DNN) and attack models can be accessed via APIs provided on the Github.

A.3.2 Hardware dependencies

No specific hardware is required to run this code.

A.3.3 Software dependencies

Our codebase can be run in the python environment using any python package manager. For ease of use, we have provided instructions on how to set up the environment using Anaconda.

A.3.4 Data sets

We use three publicly available datasets: General Social Survey (GSS), Adult, and Fivethirtyeight. We perform pre-processing on each dataset and do train-test splits (datasets available on Github). Details descriptions about each dataset can be found in Section 5.1 in the main manuscript. We provide the training datasets in our shared Github repository.

A.3.5 Models

We consider two different target models: decision tree (DT), and deepnet (DNN). We use the 'ensemble' model as the attack model. All models are trained on BigML with default features. Details about model training can be found in Section 5.2 in the main manuscript. All our ML models both target and attack, trained on each dataset, can be accessed via our provided APIs.

A.3.6 Security, privacy, and ethical concerns

Our artifact does not have security, privacy, and ethical concerns.

A.4 Installation

This artifact is dependent on the python environment. Required packages have to be installed before running the codebase. A list of requirements packages are in the *requirements* file. In our Github link, we provide step-by-step installation guidelines with Conda environment creation and installation of the dependencies. Also, procedures to run the codebase to produce outputs are all described in the GitHub readme file: https://github.com/smehnaz/black-boxMIAI

A.5 Experiment workflow

A.6 Evaluation and expected results

The following are the main claims that are supported by the artifact we submitted.

- We demonstrate two new proposed black-box model in- version attacks: (1) confidence score-based attack (CSMIA) and (2) label-only attack (LOMIA) outperforms existing FJRMIA

- Our proposed attacks can achieve better performance while estimating both binary (Table 12-13) and multi-valued (Table 10) and also multiple sensitive attributes (Table 15-16)
- We empirically show that model inversion attacks have disparate vulnerability property (Figure: 4b, 9, 10)
- We also evaluate partial knowledge attack scenarios of a target record and demonstrate that our attacks' performance is not impacted significantly in those scenarios (Figure: 5, 11-13)
- We also experiment on distributional privacy leakage and show that these attacks can also breach the privacy of datasets outside training but drawn from the same distribution. (Figure: 4a, 8)

In Sections 5.4.1, 5.4.2, and 5.4.3, we present our key comparisons of our proposed LOMIA and CSMIA attack performances compared to existing FJRMIA. This shows on different datasets, and target models our attacks outperform existing attacks in different performance metrics. To reproduce the results on LOMIA or CSMIA strategy, one has to run each attack particular strategy in our codebase described in the Github and also added end of this section. Other existing technique strategies are explained in the manuscript. In Tables 4-9, we provide target model confusion matrices and Fig. 2 in the manuscript shows the comparisons in GSS and Adult datasets. Tables 12 and 13 show performance comparisons on GSS, and Adult datasets.

For the second claim, we estimate both binary ('alcohol') and multi-valued ('age') in the FiveThirtyEight dataset (details in Section 5.4.3). We also estimate multiple attributes (inferring 'age' along with 'alcohol' (Table 16) and inferring 'alcohol' along with 'age' (Table 15)). To experiment with disparate vulnerability in model inversion attack, we query each attack model on specific subgroup instances of the training dataset, as presented in Section 5.7 of the manuscript. In the partial attack experiment, we perform the attack for estimating sensitive attributes with gradually missing more non-sensitive attributes in the training data. We present the results in Section 5.8 of the paper. We present the distributional privacy leakage experiment results in Fig. 8.

All steps for each experiment and reproduction steps are added to the Github repository. We experiment with our proposed CSMIA, LOMIA as well as baseline FJRMIA to compare performances. The different kinds of attack experiments that we perform using LOMIA, CSMIA, and FJRMIA are as follows:

- Inferring a single binary sensitive attribute
- Inferring a single multi-valued sensitive attribute
- Inferring multiple sensitive attributes
- Inferring sensitive attributes when one or more non-sensitive attributes are unknown
- Inferring sensitive attributes from data that was not originally on the training set (distributional privacy leakage)
- Analyzing disparate vulnerability of model inversion attack on different subgroups

Now we list out how these experiments' results can be reproduced one by one. One way is to use the configuration files we provided to reproduce results as a figure or a table presented in the paper. If the configuration file name is "config_x.yaml", then one only has to run the following command in the terminal "python main.py −−param config_x.yaml". Another way is to write down the configuration .yaml file and use it from the terminal in the same way.

Inferring a single binary sensitive attribute: For the Adult dataset we infer the marital attribute, and for the GSS the xmovie attribute. We use all combinations of DT and DNN models and both LOMIA and CSMIA attacks. One can use the built-in configuration files from the table in the GitHub readme file. For example: To infer marital from Adult Dataset and DT model using LOMIA attack, one can use the configuration file "configs/table_13/lomia_dt.yaml". Then one can compare the results with Table 12 and Table 13 of the paper.

Inferring a single multi-valued sensitive attribute: For the 538 dataset, we infer the multi-valued age attribute. We use the DT model and both LOMIA and CSMIA attacks. One can use the built-in configuration files from the table in the GitHub readme file. For example: To infer age using the CSMIA, one can use the configuration file "configs/table_10/csmia.yaml". Then one can compare the results with Table 10 of the paper. Because 538 is a very small dataset, in many case3 instances the target models confidence values are the same and the CSMIA chooses the sensitive attribute randomly which is the reason behind the deviation from the paper result. For LOMIA, the training of the ensemble attack model introduces the variation in the experiment result. We discuss these at the end of this section.

Inferring multiple sensitive attributes: For the 538 dataset, we infer both alcohol and age attributes. We use the DT model and both LOMIA and CSMIA attacks. One can use the built-in configuration files from the table in the GitHub readme file. For example: To infer age using the CSMIA, one can use the configuration file "configs/table_15_16/csmia.yaml". Then the results can be compared with Tables 15 and 16 from the paper. The same reason holds for the slight variation between the outputs.

Inferring sensitive attributes when one or more non-sensitive attributes are unknown: For LOMIA, we infer sensitive attributes when 1-9 non-sensitive attributes are missing in order of their importance. The details of this experiment can be found in section 5.8 of the paper. We perform the attack on both the Adult and GSS datasets

and both DT and DNN models. One can use the built-in configuration files from the table in the GitHub readme file. For example: To perform the partial knowledge attack on Adult DT, the following configuration file may be used "configs/figure_5/dt.yaml". The output can be compared with Figures 5, 11, and 12 from the paper.
For CSMIA, we infer sensitive attributes when 1-2 non-sensitive attributes are unknown. We only attack Adult DT for this setting. One can use the built-in configuration files from the directory mentioned in the GitHub readme file. For example: To perform the partial knowledge attack when occupation and capital-gain are unknown, the following configuration file may be used "configs/figure_13/occupation_capgain.yaml". The outputs can be compared with figure 13 from the paper.

Disparate Vulnerability Experiment: In this experiment, we estimate the disparate vulnerability of subgroups using the APIs on specific subgroup instances. We have to define the followings: dataset: Adult/GSS (Depends on which dataset being attacked), attack_type: *LOMIA*,*target_model_type*: DT/DNN (Depends on which target model is being attacked sensitive_attributes: ['marital']/['xmovie'] (marital for Adult, xmovie for GSS), missing_nonsensitive_attributes: [], attack_category: 'disparate_vulnerability', extra_field_for_attack_category: x (The vulnerable subgroup (one of the fields on the dataset, e.g., *male/female*)). We can also use specific built-in configuration files. For example, the following file can be used for Adult DNN sex subgroups: "configs/figure_4b/sex.yaml" One can compare the outputs with the ones presented in the paper in figure 4b, 9, and 10.

Distributional Privacy Experiment: For this experiment, a similar setup with the above code snippet can be used with attack_category: 'distributional_privacy_leakage' to get the result of dataset from the same distribution but not training data. Also, as an alternative to this, different files can be used as mentioned in the readme for this attack. For example in DNN CSMIA this can be used to get results on distributional privacy leakage: "configs/figure_8/adult_csmia_dnn_on_DSd.yaml". The outputs can be compared with Figures 4a and 8 from the paper.

LOMIA Attack Dataset Preparation: First, to build the attack dataset, one needs to query the target model. Then build the attack models with those datasets to perform the attack. We provide the attack models. Therefore, by querying the attack model, one can perform the attack. However, the attack dataset can be generated with configuration file names provided in the readme file. For example to generate dataset on Adult DT model, while inferring *marital* sensitive attribute, following configuration file can be used "configs/table_3/adult_dt.yaml".

We expect to have similar results in all experiments as presented in the empirical results. However, since the attack models are trained on a different BigML account, and BigML applies its optimization techniques while generating the attack models (ensembles), there might be a slight variation in results produced by this artifact.

A.7 Experiment customization

A.8 Notes

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.