conference

proceedings

Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium

Seattle, WA, USA

August 13–15, 2025

ISBN 978-1-939133-52-6

Sponsored by

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX

Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium

Seattle, WA, USA August 13–15, 2025

USENIX Association

Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium

August 13–15, 2025
Seattle, WA, USA

© 2025 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 978-1-939133-52-6

Artifact Evaluation Committee

Artifact Evaluation Committee Co-Chairs
Aurore Fass, *CISPA Helmholtz Center for Information Security*
Phani Vadrevu, *Louisiana State University*

Artifact Evaluation Publication Chair
Tiago Heinrich, *Max Planck Institute for Informatics*

Artifact Evaluation Committee
Tooba Aamir, *CSIRO's Data61*
Yonatan Achamyeleh, *University of California, Irvine*
Eric Ackermann, *CISPA Helmholtz Center for Information Security*
Sharad Agarwal, *University College London*
Shubham Agarwal, *CISPA Helmholtz Center for Information Security*
Ayomide Akinsanya, *Stevens Institute of Technology*
Mir Masood Ali, *University of Illinois Chicago*
Shiza Ali, *University of Washington*
Seyed Behnam Andarzian, *Radboud University*
MD Armanuzzaman, *Northeastern University*
Linard Arquint, *ETH Zurich*
Md. Ishtiaq Ashiq Khan, *Virginia Tech*
Cristian Assaiante, *Sapienza University of Rome*
Tolga Atalay, *Virginia Tech*
Jessy Ayala, *University of California, Irvine*
Muhammad Abu Bakar Aziz, *Northeastern University*
Sofiane Azogagh, *Université du Québec à Montréal*
Weiheng Bai, *University of Minnesota*
Xuesong Bai, *University of California, Irvine*
David Balash, *University of Richmond*
Luca Baldesi, *University of California, Irvine*
Leonhard Balduf, *Technische Universität Darmstadt*
Nils Bars, *CISPA Helmholtz Center for Information Security*
Fabian Bäumer, *Ruhr-Universität Bochum*
Pedro Bernardo, *TU Wien*
Yohan Beugin, *University of Wisconsin–Madison*
Evangelos Bitsikas, *Northeastern University*
Marton Bognar, *DistriNet, KU Leuven*
Halima Bouzidi, *University of California, Irvine*
Michael Brown, *Trail of Bits*
Yifeng Cai, *Peking University*
Di Cao, *Swinburne University of Technology*
Hongyan Chang, *National University of Singapore*
Jung-Woo Chang, *University of California, San Diego*
Yuanhaur Chang, *Washington University in St. Louis*
Baodong Chen, *The Ohio State University*
Bocheng Chen, *Michigan State University*
Boru Chen, *University of California, Berkeley*
Haobin Hiroki Chen, *Indiana University Bloomington*
Min Chen, *Vrije Universiteit Amsterdam*
Mingming Chen, *The Pennsylvania State University*
Xiang Chen, *The Hong Kong University of Science and Technology*
Xiaoqi Chen, *Purdue University*
Yongliang Chen, *City University of Hong Kong*
Mingfei Cheng, *Singapore Management University*
Xiaoyu Cheng, *Southeast University*
Seonyoung Cheon, *Yonsei University*
Michael Clark, *Brigham Young University*
Pascal Cotret, *Lab-STICC, ENSTA Bretagne*
Hongrui Cui, *Shanghai Jiao Tong University*
Jian Cui, *Indiana University Bloomington*
Priscilla Danso, *Stony Brook University*
Xinhao Deng, *Tsinghua University*
Zizhuang Deng, *Shandong University*
Andrea Di Dio, *Vrije Universiteit Amsterdam*
Abdulrahman Diaa, *University of Waterloo*
Guillaume Didier, *Saarland University*
Wanning Ding, *Syracuse University*
Kha Dinh, *Sungkyunkwan University*
Tian Dong, *Shanghai Jiao Tong University*
Yilu Dong, *The Pennsylvania State University*
Jialuo Du, *Tsinghua University*
Huayi Duan, *ETH Zurich*
Aoyang Fang, *The Chinese University of Hong Kong*
Tasnuva Farheen, *Louisiana State University*
Shiwei Feng, *Purdue University*
Fabian Fleischer, *Georgia Institute of Technology*
Gertjan Franken, *DistriNet, KU Leuven*
Kyle Fredrickson, *University of California, Santa Cruz*
Chuanpu Fu Fu, *Tsinghua University*
Alexander Gaidis, *Brown University*
Pierrick Gaudry, *CNRS*
Rahul George, *University of California, Riverside*
Jake Ginesin, *Northeastern University*
Gwihwan Go, *Tsinghua University*
Diksha Goel, *CSIRO's Data61*
Sanket Goutam, *Stony Brook University*
Shixuan Guan, *San Diego State University*
Keyan Guo, *University at Buffalo*
Qiming Guo, *Texas A&M University*
Xin Fan Guo, *King's College London*
Ziyi Guo, *Northwestern University*
Chakshu Gupta, *University of Twente*
Mingda Han, *Shandong University*
Alexander Hart, *CISPA Helmholtz Center for Information Security*
Sebastian Hasler, *University of Stuttgart*
Keno Hassler, *CISPA Helmholtz Center for Information Security*
Dongnan He, *Renmin University of China*
Ningyu He, *The Hong Kong Polytechnic University*
Yi He, *Wuhan University*
Adrian Herrera, *Interrupt Lab*s
Hongsheng Hu, *The University of Newcastle*
Shengtuo Hu, *Palo Alto Networks*
Long Huang, *Louisiana State University*
Xuanbo Huang, *University of Science and Technology of China*
Yibo Huang, *University of Michigan*
Yue Huang, *CSIRO's Data61*
Yujin Huang, *The University of Melbourne*
Zirui Huang, *Nanjing University*
Sander Huyghebaert, *Vrije Universiteit Brussel and KU Leuven*

Muhammad Ibrahim, *Purdue University*
Abdullah Al Ishtiaq, *The Pennsylvania State University*
Vijayanta Jain, *University of Maine*
Ismat Jarin, *University of California, Irvine*
Patrick Jattke, *ETH Zurich*
Zhengjie Ji, *Virginia Tech*
Peng Jiang, *Southeast University*
Qisheng Jiang, *Duke University*
Sindhu Reddy Kalathur Gopal, *University of Wyoming*
Rahul Kande, *Texas A&M University*
Zifeng Kang, *Johns Hopkins University*
Kelly Kaoudis, *Trail of Bits*
Sebastian Kappes, *Max Planck Institute for Informatics*
Jonas Kaspereit, *FH Münster*
Hugo Kermabon-Bobinnec, *Concordia University*
Joseph Khoury, *Louisiana State University*
Jiwon Kim, *Purdue University*
Soomin Kim, *Korea Advanced Institute of Science and Technology (KAIST)*
Torsten Krauß, *Universität Würzburg*
Zsolt Levente Kucsván, *University of Twente*
Lukas Lamster, *Graz University of Technology*
Tomer Laor, *Ben-Gurion University of the Negev*
Jaeho Lee, *Yonsei University*
Kiho Lee, *Electronics and Telecommunications Research Institute*
Yongwoo Lee, *Yonsei University*
Ahmed Lekssays, *Qatar Computing Research Institute*
Boheng Li, *Nanyang Technological University*
Kunyang Li, *University of Wisconsin–Madison*
Ruichen Li, *The Ohio State University*
Tuo Li, *Tsinghua University*
Wenhao Li, *Shandong University*
Xiang Li, *Nankai University*
Ying Li, *University of California, Los Angeles*
Zichuan Li, *Indiana University Bloomington*
Zhengyao Lin, *Carnegie Mellon University*
Zilong Lin, *Indiana University Bloomington*
Changming Liu, *Northeastern University*
Jing Liu, *University of California, Irvine, and Max Planck Institute for Security and Privacy*
Junming Liu, *Zhejiang University*
Mingyi Liu, *Georgia Institute of Technology*
Shuofeng Liu, *The University of Queensland*
Yupei Liu, *The Pennsylvania State University*
Zhengyu Liu, *Johns Hopkins University*
Zhuoran Liu, *Radboud University*
Zhipeng Lu, *University of Minnesota*
Kaixuan Luo, *The Chinese University of Hong Kong*
Mengyao Ma, *The University of Queensland*
Xiaoyue Ma, *George Mason University*
Pathum Chamikara Mahawaga Arachchige, *CSIRO's Data61*
Noah Mauthe, *CISPA Helmholtz Center for Information Security*
Muhammad Hammad Mazhar, *University of Northern Colorado*
Seyed Mohammad Mehdi Mirnajafizadeh, *Wayne State University*
Nimish Mishra, *Indian Institute of Technology Kharagpur*
Atefeh Mohseni, *University of California, Santa Barbara*
Jonas Möller, *Technische Universität Berlin*
Antonio Muñoz, *University of Málaga*
Muhammad Muzammil, *Stony Brook University*
Tao Ni, *City University of Hong Kong*
Pankaj Niroula, *College of William & Mary*
Alexandra Nisenoff, *Carnegie Mellon University*
Jean-Charles Noirot Ferrand, *University of Wisconsin–Madison*
Grigoris Ntousakis, *Brown University*
Paul L. R. Olivier, *N/A*
Chaoyuan Peng, *Zhejiang University*
Maxime Puys, *Université Clermont Auvergne*
Yiting Qu, *CISPA Helmholtz Center for Information Security*
Vaishnavi Raghavajosyula, *Max Planck Institute for Informatics*
Vidya Lakshmi Rajagopalan, *Stevens Institute of Technology*
Yidong Ren, *Michigan State University*
Huanyao Rong, *Indiana University Bloomington*
Jost Rossel, *Paderborn University*
Sina Rostami, *Max Planck Institute for Informatics*
Abhinaya S B, *North Carolina State University*
Mohammad Hammas Saeed, *George Washington University*
Gustavo Sánchez, *Karlsruhe Institute of Technology*
Christoph Sendner, *Universität Würzburg*
Xiangmin Shen, *Northwestern University*
Zilin Shen, *Purdue University*
Johnny So, *Stony Brook University*
Wei Song, *UNSW Sydney*
Aleksei Stafeev, *CISPA Helmholtz Center for Information Security*
Sneha Sudhakaran, *Florida Institute of Technology*
Elias Suvanto, *University of Luxembourg and CryptoLab Inc.*
Shahroz Tariq, *CSIRO's Data61*
Billy Tsouvalas, *Stony Brook University*
Tyler Tucker, *University of Florida*
Devon Tuma, *University of Minnesota*
Ryan Vrecenar, *Sandia National Laboratories*
Junpeng Wan, *Purdue University*
Kun Wang, *Zhejiang University*
Lingzhi Wang, *Northwestern University*
Nan Wang, *CSIRO's Data61*
Shuo Wang, *Shanghai Jiao Tong University*
Xianbo Wang, *The Chinese University of Hong Kong*
Feng Wei, *University at Buffalo*
Felix Weissberg, *Technische Universität Berlin*
Hongbo Wen, *University of California, Santa Barbara*
Annika Wilde, *Ruhr-Universität Bochum*
Cong Wu, *The University of Hong Kong*
Stanley Wu, *The University of Chicago*
Xiaoyuan Wu, *Carnegie Mellon University*
Dongwei Xiao, *The Hong Kong University of Science and Technology*
Yunming Xiao, *University of Michigan*
Qinge Xie, *Georgia Institute of Technology*
Haichuan Xu, *Georgia Institute of Technology*
Jiacheng Xu, *Zhejiang University*
Minghui Xu, *Shandong University*
Shenao Yan, *University of Connecticut*

Huanqi Yang, *City University of Hong Kong*
Tianchang Yang, *The Pennsylvania State University*
Yupeng Yang, *Georgia Institute of Technology*
Yuxiang Yang, *Tsinghua University*
Zheng Yang, *Georgia Institute of Technology*
Ziqing Yang, *CISPA Helmholtz Center for Information Security*
Mert Yassi, *Monash University*
Hosein Yavarzadeh, *University of California, San Diego*
Zahra Yazdani, *Georgia Institute of Technology*
Mingxi Ye, *Sun Yat-sen University*
Feiyang Yu, *Duke University*
Hao Yu, *National University of Defense Technology*
Jiahao Yu, *Northwestern University*
Jianjia Yu, *Johns Hopkins University*
Jiongchi Yu, *Singapore Management University*
Zheng Yu, *Northwestern University*
Zhiyuan Yu, *Washington University in St. Louis*
Ying Yuan, *University of Padua*
Gabriel Zaid, *Thales Group*
Kyle Zeng, *Arizona State University*
Bowen Zhang, *The Hong Kong University of Science and Technology*
Cen Zhang, *Nanyang Technological University*
Guangsheng Zhang, *University of Technology Sydney*
He Zhang, *RMIT University*
Junqi Zhang, *University of Science and Technology of China*
Kaiyi Zhang, *Shanghai Jiao Tong University*
Mingming Zhang, *Zhongguancun Laboratory*
Runze Zhang, *Georgia Institute of Technology*
Xin Zhang, *Peking University*
Yiming Zhang, *Tsinghua University*
Yiwei Zhang, *Purdue University*
Yunyi Zhang, *National University of Defense Technology and Tsinghua University*
Zenong Zhang, *Google*
Zidong Zhang, *Simon Fraser University*
Hanqing Zhao, *Tsinghua University*
Shixuan Zhao, *The Ohio State University*
Shenghan Zheng, *University of California, Riverside*
Xia Zhou, *Zhejiang University*
Yi Zhou, *Carnegie Mellon University*
Ting Zhu, *The Ohio State University*
Xiaochen Zhu, *Massachusetts Institute of Technology*

Message from the USENIX Security '25 Artifact Evaluation Committee Co-Chairs

On behalf of USENIX, we are pleased to welcome you to the artifact evaluation (AE) proceedings of the 34th USENIX Security Symposium. This year marks a major milestone for our community: for the first time, artifact evaluation has become a mandatory part of the conference in the scope of USENIX Security's new Open Science policy. Since its introduction in 2020, artifact evaluation has steadily grown in both scope and adoption, and this year's policy shift reflects the maturation of that process. Our collective goal remains the same: to strengthen the reproducibility, transparency, and impact of security research.

To support this major transition, we restructured the artifact evaluation process into two distinct phases. The first phase, conducted after paper acceptance, focuses on availability checks, ensuring that all artifacts referenced in accepted papers are publicly and permanently accessible online. At the end of this phase, the authors submitted the camera-ready version of the paper with the available link of the artifact. The second phase, following the final paper submission, evaluates functionality and reproducibility. This second phase is based on an artifact appendix, describing how to use the artifact, including how to reproduce the main claims from the accompanying paper. This two-phase separation allowed us to enforce the open-science requirements while still providing authors and evaluators the time and structure needed to assess artifacts thoroughly. Similar to paper submissions, the artifact evaluation process offered two submission cycles, each composed of the two phases described above.

The result was a record number of 396 artifacts formally reviewed, reflecting both the enthusiasm and the challenges of mandatory participation. We saw a wide diversity of artifacts—ranging from survey instruments to full experimental frameworks—across all security topics. Each artifact received at least two reviews for phase 1 (availability verification) and two for phase 2 (functionality and reproducibility assessments). The review process for availability verification took just a few days, and five weeks for functionality and reproducibility assessments. During this review period, authors and reviewers interacted to solve any problems, clarify some instructions, and overall improve the quality of the artifacts; this discussion phase led to more than 4,000 comments. Throughout, the Artifact Evaluation Committee (AEC) demonstrated remarkable diligence in applying the badge criteria: Artifacts Available, Artifacts Functional, and Results Reproduced. The committee's work underscores not only the growing cultural shift toward open science in security research but also the community's commitment to making reproducibility a lasting standard at USENIX Security.

To build a strong AEC, we reached out to the community, past AEC members of the major security conferences, and authors of accepted papers at USENIX Security '25, to (self-)nominate candidates. We received a record number of 414 (self-) nominations! After some screening, we selected 234 AEC members (and an additional 14 who helped for a single cycle) from across the world with varying expertise, consisting mostly of Ph.D. students, postdocs, and early career researchers from academia and industry.

During the AE process, each evaluator reviewed on average two artifacts per cycle (minimum of one, maximum of eight for AEC members who volunteered for more). In parallel to the reviews, the artifacts and appendices were automatically checked to ensure that 1) authors provided a stable reference to the stored artifacts; and 2) that the appendices were complete and formatted correctly. This helps to guarantee reproducibility of research results through reliable artifacts.

In total, 396 papers (212 in Cycle 1 and 184 in Cycle 2) participated in the mandatory availability verification, and all were awarded the Artifacts Available (AA) badge for making their research artifacts permanently accessible. Of these, 207 papers applied for the Artifacts Functional (AF) badge and 168 applied for both the AF and Results Reproduced (RR) badges. The AF evaluations had a 98.55% acceptance rate, while RR evaluations had an 87.5% acceptance rate. Ultimately, 204 papers received the AF badge, recognizing the successful execution, documentation, and completeness of their artifacts, and 150 papers earned the RR badge, demonstrating that their artifacts could be used with the provided documentation to reproduce the experimental results. Compared to USENIX Security '24, these outcomes represent a 64.29% increase in AF badges and a 51.52% increase in RR badges, underscoring the community's growing commitment to and enthusiasm for artifact evaluation.

To further promote excellence in artifact evaluation and highlight outstanding contributions, we continued the tradition of presenting awards to both authors and evaluators. Unlike last year, where a separate subcommittee selected the best artifacts, this year's Distinguished Artifact Awards were determined through votes from the entire Artifact Evaluation Committee (AEC). Six papers were selected for this honor, each earning near-unanimous praise for artifacts that reproduced all claims out of the box, included comprehensive documentation and automation, and offered responsive author support. Most importantly, these artifacts are expected to have a broad impact and inspire future research in our community.

We also introduced a more structured set of reviewer awards to recognize the invaluable work of the AEC. The Distinguished Reviewer Award (18 recipients) went to evaluators who went above and beyond by providing exceptionally detailed, constructive feedback that not only strengthened the evaluation process but, in many cases, also improved the accompanying papers. The Noteworthy Reviewer Recognition Award (44 recipients) recognized committee members who consistently delivered clear, thoughtful, and timely reviews, demonstrating reliability and engagement throughout the evaluation. Finally, the Artifact Ninja Reviewer Recognition Award honored 17 recipients (the list of awardees is available at secartifacts.github.io/usenixsec2025/awards#-ninja-reviewer-recognition) who stepped in to take on additional reviews, often at short notice, to ensure timely and thorough evaluations when unexpected challenges arose.

Together, these awards and recognitions celebrate the collective effort of the AEC and the broader community, acknowledging both the authors who set a high bar for artifact quality and the reviewers whose dedication ensured a fair, rigorous, and supportive evaluation process.

We are tremendously grateful to the AEC. The evaluators spent countless hours diligently working with the authors to help them improve their artifacts. Without this effort, the artifact availability verification of the 396 papers and this year's proceedings of 204 artifact appendices would not have been possible.

Aurore Fass, *CISPA Helmholtz Center for Information Security*
Phani Vadrevu, *Louisiana State University*
USENIX Security '25 Artifact Evaluation Committee Co-Chairs

Tiago Heinrich, *Max Planck Institute for Informatics*
USENIX Security '25 Artifact Evaluation Publication Chair

Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium

August 13–15, 2025
Seattle, WA, USA

Wednesday, August 13

Social Issues and Usable Security and Privacy

Easy As *Child's Play*: An Empirical Study on Age Verification of Adult-Oriented Android Apps 1
Yifan Yao, Shawn McCollum, Zhibo Sun, and Yue Zhang, *Drexel University*

Abusability of Automation Apps in Intimate Partner Violence 5
Shirley Zhang, *University of Wisconsin–Madison;* Paul Chung, *University of California, San Diego;* Jacob Vervelde, Nishant Korapati, Rahul Chatterjee, and Kassem Fawaz, *University of Wisconsin–Madison*

Voluntary Investment, Mandatory Minimums, or Cyber Insurance: What Minimizes Losses? 9
Adam Hastings and Simha Sethumadhavan, *Columbia University*

A First Look at Governments' Enterprise Security Guidance. 13
Kimberly Ruth and Raymond Buernor Obu, *Stanford University;* Ifeoluwa Shode, *Fisk University;* Gavin Li, *Stanford University;* Carrie Gates, *FS-ISAC;* Grant Ho, *University of Chicago;* Zakir Durumeric, *Stanford University*

Characterizing and Detecting Propaganda-Spreading Accounts on Telegram 17
Klim Kireev, *EPFL, MPI-SP Max Plank Institute for Security and Privacy;* Yevhen Mykhno, *unaffiliated;* Carmela Troncoso, *EPFL, MPI-SP Max Plank Institute for Security and Privacy;* Rebekah Overdorf, *Ruhr University Bochum (RUB), Research Center Trustworthy Data Science and Security in University Alliance Ruhr, University of Lausanne*

LLM Security and Attacks

GradEscape: A Gradient-Based Evader Against AI-Generated Text Detectors 19
Wenlong Meng, Shuguo Fan, and Chengkun Wei, *Zhejiang University;* Min Chen, *Vrije Universiteit Amsterdam;* Yuwei Li, *National University of Defense Technology and Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation;* Yuanchao Zhang, *Mybank, Ant Group;* Zhikun Zhang and Wenzhi Chen, *Zhejiang University*

HateBench: Benchmarking Hate Speech Detectors on LLM-Generated Content and Hate Campaigns. 23
Xinyue Shen, Yixin Wu, Yiting Qu, and Michael Backes, *CISPA Helmholtz Center for Information Security;* Savvas Zannettou, *Delft University of Technology;* Yang Zhang, *CISPA Helmholtz Center for Information Security*

EmbedX: Embedding-Based Cross-Trigger Backdoor Attack Against Large Language Models 27
Nan Yan and Yuqing Li, *Wuhan University;* Xiong Wang, *Huazhong University of Science and Technology;* Jing Chen and Kun He, *Wuhan University;* Bo Li, *Hong Kong University of Science and Technology*

Game of Arrows: On the (In-)Security of Weight Obfuscation for On-Device TEE-Shielded LLM Partition Algorithms. 31
Pengli Wang, *MOEKey Lab of HCST (PKU), School of Computer Science, Peking University;* Bingyou Dong, *ByteDance;* Yifeng Cai, *MOEKey Lab of HCST (PKU), School of Computer Science, Peking University;* Zheng Zhang, *ByteDance;* Junlin Liu, *MOEKey Lab of HCST (PKU), School of Computer Science, Peking University;* Huanran Xue, Ye Wu, and Yao Zhang, *ByteDance;* Ziqi Zhang, *University of Illinois Urbana-Champaign*

Refusal Is Not an Option: Unlearning Safety Alignment of Large Language Models 35
Minkyoo Song, Hanna Kim, Jaehan Kim, Seungwon Shin, and Sooel Son, *KAIST*

Software Security 1: Vulnerability Analysis, Exploit Synthesis, and Methodology

Narrowbeer: A Practical Replay Attack Against the Widevine DRM 37
Florian Roudot and Mohamed Sabt, *Univ Rennes, CNRS, IRISA*

Lancet: A Formalization Framework for Crash and Exploit Pathology. 39
Qinrun Dai, Kirby Linvill, Yueqi Chen, and Gowtham Kaki, *University of Colorado Boulder*

Synthesis of Code-Reuse Attacks from p-code Programs 43
Mark DenHoed and Tom Melham, *University of Oxford*

Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis 47
Yuxi Ling, *National Univeristy of Singapore;* Gokul Rajiv, *National University of Singapore;* Kiran Gopinathan, *University of Illinois Urbana-Champaign;* Ilya Sergey, *National University of Singapore*

My ZIP isn't your ZIP: Identifying and Exploiting Semantic Gaps Between ZIP Parsers 49
Yufan You, *Tsinghua University;* Jianjun Chen, *Tsinghua University; Zhongguancun Laboratory;* Qi Wang, *Tsinghua University;* Haixin Duan, *Tsinghua University; Zhongguancun Laboratory*

Tady: A Neural Disassembler without Structural Constraint Violations 51
Siliang Qin, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Fengrui Yang and Hao Wang, *Tsinghua University;* Bolun Zhang, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Zeyu Gao and Chao Zhang, *Tsinghua University;* Kai Chen, *Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences*

LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models 55
Ahmed Lekssays and Hamza Mouhcine, *Qatar Computing Research Institute;* Khang Tran, *New Jersey Institute of Technology;* Ting Yu, *Mohamed bin Zayed University of Artificial Intelligence;* Issa Khalil, *Qatar Computing Research Institute*

X.509DoS: Exploiting and Detecting Denial-of-Service Vulnerabilities in Cryptographic Libraries using Crafted X.509 Certificates 59
Bing Shi, Wenchao Li, Yuchen Wang, and Xiaolong Bai, *Alibaba Group;* Luyi Xing, *Indiana University Bloomington*

System Security 1: Threat Detection, Exploitation, and Adaptive Defenses

CoVault: Secure, Scalable Analytics of Personal Data 63
Roberta De Viti and Isaac Sheff, *Max Planck Institute for Software Systems* (*MPI-SWS*)*, Saarland Informatics Campus;* Noemi Glaeser, *Max Planck Institute for Security and Privacy* (*MPI-SP*) *and University of Maryland;* Baltasar Dinis, *Instituto Superior Técnico* (*ULisboa*)*, INESC-ID;* Rodrigo Rodrigues, *Instituto Superior Técnico* (*ULisboa*) / *INESC-ID;* Bobby Bhattacharjee, *University of Maryland;* Anwar Hithnawi, *ETH Zürich;* Deepak Garg and Peter Druschel, *Max Planck Institute for Software Systems* (*MPI-SWS*)*, Saarland Informatics Campus*

EvilEDR: Repurposing EDR as an Offensive Tool 67
Kotaiba Alachkar, *Delft University of Technology;* Dirk Gaastra, *Independent Researcher;* Eduardo Barbaro, Michel van Eeten, and Yury Zhauniarovich, *Delft University of Technology*

Nothing is Unreachable: Automated Synthesis of Robust Code-Reuse Gadget Chains for Arbitrary Exploitation Primitives 71
Nicolas Bailluet, *Univ Rennes, Inria, CNRS, IRISA;* Emmanuel Fleury, *Univ Bordeaux, CNRS, LaBRI;* Isabelle Puaut and Erven Rohou, *Univ Rennes, Inria, CNRS, IRISA*

From Constraints to Cracks: Constraint Semantic Inconsistencies as Vulnerability Beacons for Embedded Systems 75
Jiaxu Zhao, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences; Beijing Key Laboratory of Network Security and Protection Technology;* Yuekang Li, *University of New South Wales;* Yanyan Zou, Yang Xiao, Naijia Jiang, Yeting Li, Nanyu Zhong, Bingwei Peng, Kunpeng Jian, and Wei Huo, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences; Beijing Key Laboratory of Network Security and Protection Technology*

Network Security 1: Censorship, Evasion, and Trustworthy Infrastructure

Exposing and Circumventing SNI-based QUIC Censorship of the Great Firewall of China 77
Ali Zohaib, *University of Massachusetts Amherst;* Qiang Zao, *GFW Report;* Jackson Sippe and Abdulrahman Alaraj, *University of Colorado Boulder;* Amir Houmansadr, *University of Massachusetts Amherst;* Zakir Durumeric, *Stanford University;* Eric Wustrow, *University of Colorado Boulder*

***Ares*: Comprehensive Path Hijacking Detection via Routing Tree 81**
Yinxiang Tao, *Institute of Network Sciences and Cyberspace, Tsinghua University, Beijing, China;* Chengwan Zhang, *unaffiliated;* Changqing An, *Institute of Network Sciences and Cyberspace, Tsinghua University, Beijing, China;* Shuying Zhuang, *Zhongguancun Laboratory, Beijing, China;* Jilong Wang, *Quan Cheng Laboratory, 250103, Jinan, Shandong, China and Institute of Network Sciences and Cyberspace, Tsinghua University, Beijing, China;* Congcong Miao, *Tencent*

Trust but Verify: An Assessment of Vulnerability Tagging Services 85
Szu-Chun Huang, Harm Griffioen, Max van der Horst, Georgios Smaragdakis, Michel van Eeten, and Yury Zhauniarovich, *Delft University of Technology*

Catch-22: Uncovering Compromised Hosts using SSH Public Keys 87
Cristian Munteanu, *Max Planck Institute for Informatics;* Georgios Smaragdakis, *Delft University of Technology;* Anja Feldmann and Tobias Fiebig, *Max Planck Institute for Informatics*

ML and AI Security 1: Images

Exposing the Guardrails: Reverse-Engineering and Jailbreaking Safety Filters in DALL·E Text-to-Image Pipelines 91
Corban Villa, *New York University Abu Dhabi;* Shujaat Mirza, *New York University;* Christina Pöpper, *New York University Abu Dhabi*

Self-interpreting Adversarial Images 95
Tingwei Zhang, Collin Zhang, and John X. Morris, *Cornell Tech;* Eugene Bagdasarian, *University of Massachusetts Amherst;* Vitaly Shmatikov, *Cornell Tech*

System Security 2: Trusted and Robust Computing

TORchlight: Shedding Light on Real-World Attacks on Cloudless IoT Devices Concealed within the Tor Network. 99
Yumingzhi Pan and Zhen Ling, *Southeast University;* Yue Zhang, *Drexel University;* Hongze Wang, Guangchi Liu, and Junzhou Luo, *Southeast University;* Xinwen Fu, *University of Massachusetts Lowell*

CloudFlow: Identifying Security-sensitive Data Flows in Serverless Applications 103
Giuseppe Raffa, *Royal Holloway, University of London;* Jorge Blasco, *Universidad Politécnica de Madrid;* Dan O'Keeffe, *Royal Holloway, University of London;* Santanu Kumar Dash, *University of Surrey*

Serverless Functions Made Confidential and Efficient with Split Containers 107
Jiacheng Shi, Jinyu Gu, Yubin Xia, and Haibo Chen, *Shanghai Jiao Tong University*

Exploring and Exploiting the Resource Isolation Attack Surface of WebAssembly Containers 109
Zhaofeng Yu, Dongyang Zhan, Lin Ye, Haining Yu, and Hongli Zhang, *Harbin Institute of Technology;* Zhihong Tian, *Guangzhou University*

Transparent Attested DNS for Confidential Computing Services. 113
Antoine Delignat-Lavaud, Cédric Fournet, Kapil Vaswani, Manuel Costa, and Sylvan Clebsch, *Azure Research, Microsoft;* Christoph M. Wintersteiger, *Imandra*

TLBlur: Compiler-Assisted Automated Hardening against Controlled Channels on Off-the-Shelf Intel SGX Platforms. 117
Daan Vanoverloop, *DistriNet, KU Leuven;* Andrés Sánchez, *EPFL, Amazon;* Flavio Toffalini, *EPFL, RUB;* Frank Piessens, *DistriNet, KU Leuven;* Mathias Payer, *EPFL;* Jo Van Bulck, *DistriNet, KU Leuven*

Blockchain Security, Attacks, and Defenses

Available Attestation: Towards a Reorg-Resilient Solution for Ethereum Proof-of-Stake 121
Mingfei Zhang, *Shandong University;* Rujia Li, *Tsinghua University;* Xueqian Lu, *Independent Reseacher;* Sisi Duan, *Tsinghua University*

Voting-Bloc Entropy: A New Metric for DAO Decentralization 125
Andres Fabrega, *Cornell University;* Amy Zhao, *IC3;* Jay Yu, *Stanford University;* James Austgen, *Cornell Tech;* Sarah Allen, *IC3 and Flashbots;* Kushal Babel, *Cornell Tech and IC3;* Mahimna Kelkar, *Cornell Tech;* Ari Juels, *Cornell Tech and IC3*

Let's Move2EVM **129**
Lorenzo Benetollo, *Ca' Foscari University of Venice, University of Camerino, and Christian Doppler Laboratory Blockchain Technologies for the Internet of Things;* Andreas Lackner, *TU Wien;* Matteo Maffei and Markus Scherer, *TU Wien and Christian Doppler Laboratory Blockchain Technologies for the Internet of Things*

Surviving in Dark Forest: Towards Evading the Attacks from Front-Running Bots in Application Layer **133**
Zuchao Ma, Muhui Jiang, Feng Luo, and Xiapu Luo, *The Hong Kong Polytechnic University;* Yajin Zhou, *Zhejiang University*

LLM Privacy

Unlocking the Power of Differentially Private Zeroth-order Optimization for Fine-tuning LLMs. **135**
Ergute Bao, *Alibaba Group;* Yangfan Jiang, *National University of Singapore;* Fei Wei, *Alibaba Group;* Xiaokui Xiao, *National University of Singapore;* Zitao Li, Yaliang Li, and Bolin Ding, *Alibaba Group*

Evaluating LLM-based Personal Information Extraction and Countermeasures. **139**
Yupei Liu, *The Pennsylvania State University;* Yuqi Jia, *Duke University;* Jinyuan Jia, *The Pennsylvania State University;* Neil Zhenqiang Gong, *Duke University*

Embedded and Hardware Security

Enabling Low-Cost Secure Computing on Untrusted In-Memory Architectures **141**
Sahar Ghoflsaz Ghinani, Jingyao Zhang, and Elaheh Sadredini, *University of California, Riverside*

Software Availability Protection in Cyber-Physical Systems **143**
Ao Li, Jinwen Wang, and Ning Zhang, *Washington University in St. Louis*

GDMA: Fully Automated DMA Rehosting via Iterative Type Overlays **147**
Tobias Scharnowski, Simeon Hoffmann, Moritz Bley, and Simon Wörner, *CISPA Helmholtz Center for Information Security;* Daniel Klischies, *Ruhr-Universität Bochum;* Felix Buchmann, Nils Ole Tippenhauer, and Thorsten Holz, *CISPA Helmholtz Center for Information Security;* Marius Muench, *University of Birmingham*

KINTSUGI: Secure Hotpatching for Code-Shadowing Real-Time Embedded Systems **153**
Philipp Mackensen, *Ruhr University Bochum;* Christian Niesler, *University of Duisburg-Essen;* Roberto Blanco, *Eindhoven University of Technology/MPI-SP;* Lucas Davi, *University of Duisburg-Essen;* Veelasha Moonsamy, *Ruhr University Bochum*

Crypto 1: Zero Knowledge and Multi-Party Computation

Dumbo-MPC: Efficient Fully Asynchronous MPC with Optimal Resilience **157**
Yuan Su, *Xi'an Jiaotong University;* Yuan Lu, *Institute of Software Chinese Academy of Sciences;* Jiliang Li, *Xi'an Jiaotong University;* Yuyi Wang, *CRRC Zhuzhou Institute;* Chengyi Dong, *Xi'an Jiaotong University;* Qiang Tang, *The University of Sydney*

FABLE: Batched Evaluation on Confidential Lookup Tables in 2PC **161**
Zhengyuan Su, *Tsinghua University;* Qi Pang, Simon Beyzerov, and Wenting Zheng, *Carnegie Mellon University*

MAESTRO: Multi-Party AES Using Lookup Tables **165**
Hiraku Morita, *Aarhus University and University of Copenhagen;* Erik Pohle, *COSIC, KU Leuven;* Kunihiko Sadakane, *The University of Tokyo;* Peter Scholl, *Aarhus University;* Kazunari Tozawa, *The University of Tokyo;* Daniel Tschudi, *Concordium and Eastern Switzerland University of Applied Sciences* (*OST*)

Efficient Multi-Party Private Set Union Without Non-Collusion Assumptions **169**
Minglang Dong, *School of Cyber Science and Technology, Shandong University; Quan Cheng Laboratory; Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University;* Cong Zhang, *Institute for Advanced Study, BNRist, Tsinghua University;* Yujie Bai and Yu Chen, *School of Cyber Science and Technology, Shandong University; Quan Cheng Laboratory; Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University*

Scalable Collaborative zk-SNARK and Its Application to Fully Distributed Proof Delegation **173**
Xuanming Liu, Zhelei Zhou, and Yinghao Wang, *Zhejiang University;* Yanxin Pang, *Tsinghua University;* Jinye He, *University of Virginia;* Bingsheng Zhang and Xiaohu Yang, *Zhejiang University;* Jiaheng Zhang, *National University of Singapore*

DFS: Delegation-friendly zkSNARK and Private Delegation of Provers 177
Yuncong Hu, *Shanghai Jiao Tong University;* Pratyush Mishra, *University of Pennsylvania;* Xiao Wang, *Northwestern University;* Jie Xie, *Shanghai Jiao Tong University;* Kang Yang, *State Key Laboratory of Cryptology;* Yu Yu, *Shanghai Jiao Tong University and Shanghai Qi Zhi Institute;* Yuwen Zhang, *University of California, Berkeley*

SoK: Understanding zk-SNARKs: The Gap Between Research and Practice 179
Junkai Liang and Daqi Hu, *Peking University;* Pengfei Wu, *Singapore Management University;* Yunbo Yang, *East China Normal University;* Qingni Shen and Zhonghai Wu, *Peking University*

Thursday, August 14

Usable Privacy and Security 2: Software and Experts

A limited technical background is sufficient for attack-defense tree acceptability 183
Nathan Daniel Schiele and Olga Gadyatskaya, *Leiden University*

LLM Security 2: Jailbreaking and Prompt Stealing

Prompt Obfuscation for Large Language Models 187
David Pape and Sina Mavali, *CISPA Helmholtz Center for Information Security;* Thorsten Eisenhofer, *Berlin Institute for the Foundations of Learning and Data* (*BIFOLD*) *and Technische Universität Berlin;* Lea Schönherr, *CISPA Helmholtz Center for Information Security*

TwinBreak: Jailbreaking LLM Security Alignments based on Twin Prompts 191
Torsten Krauß, Hamid Dashtbani, and Alexandra Dmitrienko, *University of Würzburg*

Hardware Security 1: Microarchitectures

SoK: So, You Think You Know All About Secure Randomized Caches? 197
Anubhav Bhatla, Hari Rohit Bhavsar, Sayandeep Saha, and Biswabandan Panda, *Indian Institute of Technology Bombay*

TEEcorrelate: An Information-Preserving Defense against Performance-Counter Attacks on TEEs 201
Hannes Weissteiner and Fabian Rauscher, *Graz University of Technology;* Robin Leander Schröder, *Fraunhofer SIT and Fraunhofer Austria;* Jonas Juffinger and Stefan Gast, *Graz University of Technology;* Jan Wichelmann and Thomas Eisenbarth, *University Luebeck;* Daniel Gruss, *Graz University of Technology*

Systematic Evaluation of Randomized Cache Designs against Cache Occupancy 203
Anirban Chakraborty, *Max Planck Institute for Security and Privacy;* Nimish Mishra, *Indian Institute of Technology Kharagpur;* Sayandeep Saha, *Indian Institute of Technology Bombay;* Sarani Bhattacharya and Debdeep Mukhopadhyay, *Indian Institute of Technology Kharagpur*

Exploiting Inaccurate Branch History in Side-Channel Attacks 207
Yuhui Zhu, *Scuola Superiore Sant'Anna and Scuola IMT Alti Studi Lucca;* Alessandro Biondi, *Scuola Superiore Sant'Anna*

Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks 211
Alvise de Faveri Tron, Raphael Isemann, Hany Ragab, Cristiano Giuffrida, Klaus von Gleissenthall, and Herbert Bos, *Vrije Universiteit Amsterdam*

Place Protections at the Right Place: Targeted Hardening for Cryptographic Code against Spectre v1 215
Yiming Zhu, Wenchao Huang, and Yan Xiong, *University of Science and Technology of China*

Encarsia: Evaluating CPU Fuzzers via Automatic Bug Injection 219
Matej Bölcskei, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi, *ETH Zurich*

Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions 223
Sandro Rüegge, Johannes Wikner, and Kaveh Razavi, *ETH Zurich*

Privacy 1: Differential Privacy and Audit

GraphAce: Secure Two-Party Graph Analysis Achieving Communication Efficiency 227
Jiping Yu, *Tsinghua University and Ant Group;* Kun Chen, *Ant Group;* Yunyi Chen and Xiaoyu Fan, *Tsinghua University and Ant Group;* Xiaowei Zhu and Cheng Hong, *Ant Group;* Wenguang Chen, *Tsinghua University and Ant Group*

HawkEye: Statically and Accurately Profiling the Communication Cost of Models in Multi-party Learning 231
Wenqiang Ruan, Xin Lin, Ruisheng Zhou, and Guopeng Lin, *Fudan University;* Shui Yu, *University of Technology Sydney;* Weili Han, *Fudan University*

General-Purpose *f*-DP Estimation and Auditing in a Black-Box Setting 235
Önder Askin, Holger Dette, and Martin Dunsche, *Ruhr-University Bochum;* Tim Kutta, *Aarhus University;* Yun Lu, *University of Victoria;* Yu Wei and Vassilis Zikas, *Georgia Institute of Technology*

Addressing Sensitivity Distinction in Local Differential Privacy: A General Utility-Optimized Framework 239
Xingyu He, Youwen Zhu, and Rongke Liu, *Nanjing University of Aeronautics and Astronautics;* Gaoning Pan, *Hangzhou Dianzi University;* Changyu Dong, *Guangzhou University*

Further Study on Frequency Estimation under Local Differential Privacy 243
Huiyu Fang, Liquan Chen, and Suhui Liu, *Southeast University*

Software Security and Usable Security

Stack Overflow Meets Replication: Security Research Amid Evolving Code Snippets 245
Alfusainey Jallow, *CISPA Helmholtz Center for Information Security and Saarland University;* Sven Bugiel, *CISPA Helmholtz Center for Information Security*

Beyond Exploit Scanning: A Functional Change-Driven Approach to Remote Software Version Identification ... 251
Jinsong Chen, Mengying Wu, Geng Hong, and Baichao An, *Fudan University;* Mingxuan Liu, *Zhongguancun Laboratory;* Lei Zhang, *Fudan University;* Baojun Liu, *Tsinghua University;* Haixin Duan, *Tsinghua University and Quancheng Laboratory;* Min Yang, *Fudan University*

"I'm trying to learn. . . and I'm shooting myself in the foot": Beginners' Struggles When Solving Binary Exploitation Exercises 255
James Mattei, Christopher Pellegrini, and Matthew Soto, *Tufts University;* Marina Sanusi Bohuk, *MetaCTF;* Daniel Votipka, *Tufts University*

***"I have no idea how to make it safer"*: Studying Security and Privacy Mindsets of Browser Extension Developers ... 257**
Shubham Agarwal and Rafael Mrowczynski, *CISPA Helmholtz Center for Information Security;* Maria Hellenthal, *CISPA Helmholtz Centre for Information Security;* Ben Stock, *CISPA Helmholtz Center for Information Security*

Precise and Effective Gadget Chain Mining through Deserialization Guided Call Graph Construction..... 261
Yiheng Zhang, Ming Wen, and Shunjie Liu, *Huazhong University of Science and Technology;* Dongjie He, *Chongqing University;* Hai Jin, *Huazhong University of Science and Technology*

ML and AI Privacy 1: Federated Learning and Protecting Data

Aion: Robust and Efficient Multi-Round Single-Mask Secure Aggregation Against Malicious Participants 265
Yizhong Liu, Zixiao Jia, Zian Jin, Xiao Chen, Song Bian, Runhua Xu, Dawei Li, and Jianwei Liu, *Beihang University;* Yuan Lu, *Institute of Software, Chinese Academy of Sciences*

Web and Mobile Security

Doubly Dangerous: Evading Phishing Reporting Systems by Leveraging Email Tracking Techniques..... 269
Anish Chand, *Louisiana State University;* Nick Nikiforakis, *Stony Brook University;* Phani Vadrevu, *Louisiana State University*

Evaluating the Effectiveness and Robustness of Visual Similarity-based Phishing Detection Models 273
Fujiao Ji and Kiho Lee, *University of Tennessee, Knoxville;* Hyungjoon Koo, *Sungkyunkwan University;* Wenhao You and Euijin Choo, *University of Alberta;* Hyoungshick Kim, *Sungkyunkwan University;* Doowon Kim, *University of Tennessee, Knoxville*

Hercules Droidot and the murder on the JNI Express 277
Luca Di Bartolomeo and Philipp Mao, *EPFL;* Yu-Jye Tung and Jessy Ayala, *University of California, Irvine;* Samuele Doria, *University of Padua;* Paolo Celada and Marcel Busch, *EPFL;* Joshua Garcia, *University of California, Irvine;* Eleonora Losiouk, *University of Padova;* Mathias Payer, *EPFL*

No Way to Sign Out? Unpacking Non-Compliance with Google Play's App Account Deletion Requirements..... 281
Jingwen Yan, *Clemson University;* Song Liao, *Texas Tech University;* Jin Ma and Mohammed Aldeen, *Clemson University;* Salish Kumar, *Texas Tech University;* Long Cheng, *Clemson University*

Lost in the Mists of Time: Expirations in DNS Footprints of Mobile Apps 285
Johnny So, *Stony Brook University;* Iskander Sanchez-Rola, *Norton Research Group;* Nick Nikiforakis, *Stony Brook University*

TapTrap: Animation-Driven Tapjacking on Android. 291
Philipp Beer and Marco Squarcina, *TU Wien;* Sebastian Roth, *University of Bayreuth;* Martina Lindorfer, *TU Wien*

Crypto 2: Private Information Retrieval and Computation

Distributional Private Information Retrieval 295
Ryan Lehmkuhl, Alexandra Henzinger, and Henry Corrigan-Gibbs, *MIT*

Practical Keyword Private Information Retrieval from Key-to-Index Mappings 297
Meng Hao, *School of Computing & Information Systems, Singapore Management University;* Weiran Liu and Liqiang Peng, *Alibaba Group;* Cong Zhang, *Institute for Advanced Study, BNRist, Tsinghua University;* Pengfei Wu, *School of Computing & Information Systems, Singapore Management University;* Lei Zhang, *Alibaba Group;* Hongwei Li, *Peng Cheng Laboratory;* Robert H. Deng, *School of Computing & Information Systems, Singapore Management University*

Fast Enhanced Private Set Union in the Balanced and Unbalanced Scenarios 301
Binbin Tu and Yujie Bai, *School of Cyber Science and Technology, Shandong University; Quan Cheng Laboratory; Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University;* Cong Zhang, *Institute for Advanced Study, BNRist, Tsinghua University;* Yang Cao and Yu Chen, *School of Cyber Science and Technology, Shandong University; Quan Cheng Laboratory; Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University*

Network Security 2: Routing and DoS

Your Shield is My Sword: A Persistent Denial-of-Service Attack via the Reuse of Unvalidated Caches in DNSSEC Validation 305
Shuhan Zhang, *Tsinghua University and Tsinghua Shenzhen International Graduate School;* Shuai Wang and Li Chen, *Zhongguancun Laboratory;* Dan Li and Baojun Liu, *Tsinghua University*

BGP Vortex: Update Message Floods Can Create Internet Instabilities 309
Felix Stöger, *ETH Zurich;* Henry Birge-Lee, *Princeton University;* Giacomo Giuliari, *Mysten Labs;* Jordi Subira-Nieto and Adrian Perrig, *ETH Zurich*

LLM Security 3: Hallucinations, RAG Poisoning, and Agentic Attacks

Are CAPTCHAs Still Bot-hard? Generalized Visual CAPTCHA Solving with Agentic Vision Language Model . . . 313
Xiwen Teoh, *Shanghai Jiao Tong University; National University of Singapore;* Yun Lin, *Shanghai Jiao Tong University;* Siqi Li and Ruofan Liu, *National University of Singapore;* Avi Sollomoni and Yaniv Harel, *Tel Aviv University;* Jin Song Dong, *National University of Singapore*

Topic-FlipRAG: Topic-Orientated Adversarial Opinion Manipulation Attacks to Retrieval-Augmented Generation Models 317
Yuyang Gong, Zhuo Chen, Jiawei Liu, Miaokun Chen, Fengchang Yu, and Wei Lu, *Wuhan University;* XiaoFeng Wang, *Nanyang Technological University;* Xiaozhong Liu, *Worcester Polytechnic Institute*

Hardware Security 2: Signals, Waves, and Side-Channel Secrets

Sound of Interference: Electromagnetic Eavesdropping Attack on Digital Microphones Using Pulse Density Modulation. 321
Arifu Onishi, *The University of Electro-Communications;* S. Hrushikesh Bhupathiraju, Rishikesh Bhatt, and Sara Rampazzi, *University of Florida;* Takeshi Sugawara, *The University of Electro-Communications*

Blockchain Security 2: Infrastructure, Protocol Design, and Governance

CollisionRepair: First-Aid and Automated Patching for Storage Collision Vulnerabilities in Smart Contracts 325
Yu Pan and Wanjing Han, *University of Utah;* Yue Duan, *Singapore Management University;* Mu Zhang, *University of Utah*

Does Finality Gadget Finalize Your Block? A Case Study of Binance Consensus 327
Rujia Li, *Tsinghua University;* Jingyuan Ding, *Shandong University;* Qin Wang, *CSIRO Data61;* Keting Jia, *Tsinghua University;* Haibin Zhang, *Yangtze Delta Region Institute of Tsinghua University;* Sisi Duan, *Tsinghua University*

Recover from Excessive Faults in Partially-Synchronous BFT SMR 331
Tiantian Gong and Gustavo Franco Camilo, *Purdue University;* Kartik Nayak, *Duke University;* Andrew Lewis-Pye, *London School of Economics;* Aniket Kate, *Purdue University and Supra Research*

System Security 3: Mobile Platforms

NASS: Fuzzing All Native Android System Services with Interface Awareness and Coverage 335
Philipp Mao, Marcel Busch, and Mathias Payer, *EPFL*

Ariadne: Navigating through the Labyrinth of Data-Driven Customization Inconsistencies in Android 339
Parjanya Vyas, Haseeb Ur Rehman Faheem, Yousra Aafer, and N. Asokan, *University of Waterloo*

Harness: Transparent and Lightweight Protection of Vehicle Control on Untrusted Android Automotive Operating System 343
Haochen Gong, Siyu Hong, Shenyi Yang, Rui Chang, Wenbo Shen, Ziqi Yuan, Chenyang Yu, and Yajin Zhou, *Zhejiang University*

Scoop: Mitigation of Recapture Attacks on Provenance-Based Media Authentication 347
Yuxin (Myles) Liu, Habiba Farrukh, and Ardalan Amiri Sani, *UC Irvine;* Sharad Agarwal, *Microsoft;* Gene Tsudik, *UC Irvine*

Chimera: Creating Digitally Signed Fake Photos by Fooling Image Recapture and Deepfake Detectors 351
Seongbin Park, Alexander Vilesov, Jinghuai Zhang, Hossein Khalili, Yuan Tian, Achuta Kadambi, and Nader Sehatbakhsh, *University of California, Los Angeles*

***Tracking You from a Thousand Miles Away!* Turning a Bluetooth Device into an Apple AirTag Without Root Privileges 353**
Junming Chen, Xiaoyue Ma, Lannan Luo, and Qiang Zeng, *George Mason University*

Software Security 2: Patching and Repair

Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis 357
Youngjoon Kim and Sunguk Shin, *Korea University;* Hyoungshick Kim, *Sungkyunkwan University;* Jiwon Yoon, *Korea University*

APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching 361
Yu Nong, *University at Buffalo;* Haoran Yang, *Washington State University;* Long Cheng, *Clemson University;* Hongxin Hu and Haipeng Cai, *University at Buffalo*

RangeSanitizer: Detecting Memory Errors with Efficient Range Checks 363
Floris Gorter and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

Attacker Control and Bug Prioritization 365
Guilhem Lacombe and Sébastien Bardin, *Université Paris-Saclay, CEA, List, France*

ML and AI Security 2

SafeSpeech: Robust and Universal Voice Protection Against Malicious Speech Synthesis 371
Zhisheng Zhang, *Beijing University of Posts and Telecommunications;* Derui Wang, *CSIRO's Data61;* Qianyi Yang, Pengyang Huang, and Junhan Pu, *Beijing University of Posts and Telecommunications;* Yuxin Cao, *National University of Singapore;* Kai Ye, *The University of Hong Kong;* Jie Hao and Yixian Yang, *Beijing University of Posts and Telecommunications*

SoK: Automated TTP Extraction from CTI Reports – Are We There Yet? 375
Marvin Büchel, *Carl von Ossietzky Universität Oldenburg;* Tommaso Paladini, *Politecnico di Milano, NEC Laboratories Europe GmbH;* Stefano Longari, Michele Carminati, and Stefano Zanero, *Politecnico di Milano;* Hodaya Binyamini, Gal Engelberg, and Dan Klein, *Accenture Labs;* Giancarlo Guizzardi, *University of Twente;* Marco Caselli, *Siemens AG;* Andrea Continella and Maarten van Steen, *University of Twente;* Andreas Peter, *Carl von Ossietzky Universität Oldenburg;* Thijs van Ede, *University of Twente*

Whispering Under the Eaves: Protecting User Privacy Against Commercial and LLM-powered Automatic Speech Recognition Systems 379
Weifei Jin, *Beijing University of Posts and Telecommunications;* Yuxin Cao, *National University of Singapore;* Junjie Su, *Beijing University of Posts and Telecommunications;* Derui Wang, *CSIRO's Data61;* Yedi Zhang, *National University of Singapore;* Minhui Xue, *CSIRO's Data61;* Jie Hao, *Beijing University of Posts and Telecommunications;* Jin Song Dong, *National University of Singapore;* Yixian Yang, *Beijing University of Posts and Telecommunications*

When Translators Refuse to Translate: A Novel Attack to Speech Translation Systems 383
Haolin Wu, *Wuhan University and Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, China;* Chang Liu, *University of Science and Technology of China;* Jing Chen, Ruiying Du, Kun He, and Yu Zhang, *Wuhan University and Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, China;* Cong Wu and Tianwei Zhang, *Nanyang Technological University;* Qing Guo and Jie Zhang, *CFAR and IHPC, A*STAR, Singapore*

Fraud, Malware, Spam

VAPD: An Anomaly Detection Model for PDF Malware Forensics with Adversarial Robustness 385
Side Liu, *Wuhan University;* Jiang Ming, *Tulane University;* Yilin Zhou, Jianming Fu, and Guojun Peng, *Wuhan University*

High Stakes, Low Certainty: Evaluating the Efficacy of High-Level Indicators of Compromise in Ransomware Attribution 389
Max van der Horst, *Delft University of Technology;* Ricky Kho, *Sogeti;* Olga Gadyatskaya, *Leiden University;* Michel Mollema, *Northwave Cybersecurity;* Michel Van Eeten and Yury Zhauniarovich, *Delft University of Technology*

Crypto 3: Formal Methods and Private Computation

A Tale of Two Worlds, a Formal Story of WireGuard Hybridization 393
Pascal Lafourcade and Dhekra Mahmoud, *Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Mines Saint-Etienne, LIMOS, 63000 Clermont-Ferrand, France;* Sylvain Ruhault and Abdul Rahman Taleb, *Agence Nationale de la Sécurité des Systèmes d'Information* (*ANSSI*), *France*

A Formal Analysis of Apple's iMessage PQ3 Protocol 397
Felix Linker, Ralf Sasse, and David Basin, *ETH Zurich*

Towards Practical, End-to-End Formally Verified X.509 Certificate Validators with Verdict 399
Zhengyao Lin, Michael McLoughlin, and Pratap Singh, *Carnegie Mellon University;* Rory Brennan-Jones, *University of Rochester;* Paul Hitchcox, *Carnegie Mellon University;* Joshua Gancher, *Northeastern University;* Bryan Parno, *Carnegie Mellon University*

Picachv: Formally Verified Data Use Policy Enforcement for Secure Data Analytics 403
Haobin Hiroki Chen and Hongbo Chen, *Indiana University Bloomington;* Mingshen Sun, *Independent Researcher;* Chenghong Wang, *Indiana University Bloomington;* XiaoFeng Wang, *Nanyang Technological University*

OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries 407
Pratap Singh, *Carnegie Mellon University;* Joshua Gancher, *Northeastern University;* Bryan Parno, *Carnegie Mellon University*

Friday, August 15

Social Issues and Security

Vulnerability of Text-Matching in ML/AI Conference Reviewer Assignments to Collusions. 411
Jhih-Yi (Janet) Hsieh, Aditi Raghunathan, and Nihar B. Shah, *Carnegie Mellon University*

SoK: Machine Learning for Misinformation Detection 415
Madelyne Xiao and Jonathan Mayer, *Princeton University*

Network Security 3: BLE and Cellular

CoreCrisis: Threat-Guided and Context-Aware Iterative Learning and Fuzzing of 5G Core Networks. 417
Yilu Dong, Tianchang Yang, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Ali Ranjbar, Kai Tu, Tianwei Wu, Md Sultan Mahmud, and Syed Rafiul Hussain, *The Pennsylvania State University*

Gotta Detect 'Em All: Fake Base Station and Multi-Step Attack Detection in Cellular Networks 419
Kazi Samin Mubasshir, Imtiaz Karim, and Elisa Bertino, *Purdue University*

SNI5GECT: A Practical Approach to Inject aNRchy into 5G NR 425
Shijie Luo, Matheus Garbelini, Sudipta Chattopadhyay, and Jianying Zhou, *Singapore University of Technology and Design*

ML and AI Privacy 2

Disparate Privacy Vulnerability: Targeted Attribute Inference Attacks and Defenses 441
Ehsanul Kabir, Lucas Craig, and Shagufta Mehnaz, *Pennsylvania State University*

Enhanced Label-Only Membership Inference Attacks with Fewer Queries 445
Hao Li, *Institute of Software, Chinese Academy of Sciences;* Zheng Li, *Shandong University;* Siyuan Wu, Yutong Ye, Min Zhang, and Dengguo Feng, *Institute of Software, Chinese Academy of Sciences;* Yang Zhang, *CISPA Helmholtz Center for Information Security*

For Human Ears Only: Preventing Automated Monitoring on Voice Data 449
Irtaza Shahid and Nirupam Roy, *University of Maryland, College Park*

Hardware Security 3: Side-Channel and Fault Injection Attacks

MCSEE: Evaluating Advanced Rowhammer Attacks and Defenses via Automated DRAM Traffic Analysis. 453
Patrick Jattke and Michele Marazzi, *ETH Zurich;* Flavien Solt, *UC Berkeley;* Max Wipfli, Stefan Gloor, and Kaveh Razavi, *ETH Zurich*

Not so Refreshing: Attacking GPUs using RFM Rowhammer Mitigation 457
Ravan Nazaraliyev and Yicheng Zhang, *University of California, Riverside;* Sankha Baran Dutta, *Brookhaven National Laboratory;* Andres Marquez and Kevin Barker, *Pacific Northwest National Laboratory;* Nael Abu-Ghazaleh, *University of California, Riverside*

Posthammer: Pervasive Browser-based Rowhammer Attacks with Postponed Refresh Commands 461
Finn de Ridder, Patrick Jattke, and Kaveh Razavi, *ETH Zurich*

Relocate-Vote: Using Sparsity Information to Exploit Ciphertext Side-Channels. 463
Yuqin Yan, *University of Toronto;* Wei Huang, *University of Toronto and Seneca Polytechnic;* Ilya Grishchenko and Gururaj Saileshwar, *University of Toronto;* Aastha Mehta, *University of British Columbia;* David Lie, *University of Toronto*

GPUHammer: Rowhammer Attacks on GPU Memories are Practical 467
Chris S. Lin, Joyce Qu, and Gururaj Saileshwar, *University of Toronto*

SCASE: Automated Secret Recovery via Side-Channel-Assisted Symbolic Execution 471
Daniel Weber, Lukas Gerlach, and Leon Trampert, *CISPA Helmholtz Center for Information Security;* Youheng Lü, *SCHUTZWERK GmbH;* Jo Van Bulck, *DistriNet, KU Leuven;* Michael Schwarz, *CISPA Helmholtz Center for Information Security*

Shadows in Cipher Spaces: Exploiting Tweak Repetition in Hardware Memory Encryption 475
Wei Peng, Yinshuai Li, and Yinqian Zhang, *Southern University of Science and Technology*

Privacy 2: Consent, Compliance, and Provable Privacy

Websites' Global Privacy Control Compliance at Scale and over Time 479
Katherine Hausladen, Oliver Wang, and Sophie Eng, *Wesleyan University;* Jocelyn Wang, *Princeton University;* Francisca Wijaya, Matthew May, and Sebastian Zimmeck, *Wesleyan University*

ATKSCOPES: Multiresolution Adversarial Perturbation as a Unified Attack on Perceptual Hashing and Beyond 483
Yushu Zhang, Yuanyuan Sun, and Shuren Qi, *Nanjing University of Aeronautics and Astronautics;* Zhongyun Hua, *Harbin Institute of Technology, Shenzhen;* Wenying Wen and Yuming Fang, *Jiangxi University of Finance and Economics*

Usable Privacy and Security 3

How Transparent is Usable Privacy and Security Research? A Meta-Study on Current Research Transparency Practices 487
Jan H. Klemmer, Juliane Schmüser, Fabian Fischer, Jacques Suray, Jan-Ulrich Holtgrave, and Simon Lenau, *CISPA Helmholtz Center for Information Security;* Byron M. Lowens, *Indiana University Indianapolis;* Florian Schaub, *University of Michigan;* Sascha Fahl, *CISPA Helmholtz Center for Information Security*

"Helps me Take the Post With a Grain of Salt:" Soft Moderation Effects on Accuracy Perceptions and Sharing Intentions of Inauthentic Political Content on X **489**
Filipo Sharevski, *DePaul University;* Verena Distler, *Aalto University;* Florian Alt, *Ludwig Maximilians Universität, University of the Bundeswehr Munich*

Software Security 3: Fuzzing

Robust, Efficient, and Widely Available Greybox Fuzzing for COTS Binaries with System Call Pattern Feedback **493**
Jifan Xiao, *Key Laboratory of High Confidence Software Technologies, Peking University;* Peng Jiang, *Southeast University;* Zixi Zhao, Ruizhe Huang, Junlin Liu, and Ding Li, *Key Laboratory of High Confidence Software Technologies, Peking University*

BLuEMan: A Stateful Simulation-based Fuzzing Framework for Open-Source RTOS Bluetooth Low Energy Protocol Stacks **495**
Wei-Che Kao, Yen-Chia Chen, Yu-Sheng Lin, Yu-Cheng Yang, Chi-Yu Li, and Chun-Ying Huang, *National Yang Ming Chiao Tung University*

ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space **499**
Chuyang Chen, *The Ohio State University;* Brendan Dolan-Gavitt, *New York University;* Zhiqiang Lin, *The Ohio State University*

ML and AI Security 3: Backdoors, Poisoning, Unlearning

Rowhammer-Based Trojan Injection: One Bit Flip Is Sufficient for Backdooring DNNs **503**
Xiang Li, Ying Meng, Junming Chen, Lannan Luo, and Qiang Zeng, *George Mason University*

From Purity to Peril: Backdooring Merged Models From "Harmless" Benign Components **507**
Lijin Wang, *The Hong Kong University of Science and Technology* (*Guangzhou*)*;* Jingjing Wang, *Zhejiang University;* Tianshuo Cong, *Tsinghua University;* Xinlei He, *The Hong Kong University of Science and Technology* (*Guangzhou*)*;* Zhan Qin, *Zhejiang University;* Xinyi Huang, *Jinan University*

Revisiting Training-Inference Trigger Intensity in Backdoor Attacks **509**
Chenhao Lin, Chenyang Zhao, Shiwei Wang, Longtian Wang, Chao Shen, and Zhengyu Zhao, *Xi'an Jiaotong University*

Privacy 3: Attacks

HyTrack: Resurrectable and Persistent Tracking Across Android Apps and the Web **513**
Malte Wessels, Simon Koch, Jan Drescher, Louis Bettels, David Klein, and Martin Johns, *TU Braunschweig*

Seeing Through: Analyzing and Attacking Virtual Backgrounds in Video Calls **517**
Felix Weissberg, *BIFOLD & TU Berlin;* Jan Malte Hilgefort and Steve Grogorick, *TU Braunschweig;* Daniel Arp, *TU Wien;* Thorsten Eisenhofer, *BIFOLD & TU Berlin;* Martin Eisemann, *TU Braunschweig;* Konrad Rieck, *BIFOLD & TU Berlin*

Endangered Privacy: Large-Scale Monitoring of Video Streaming Services **519**
Martin Björklund and Romaric Duvignau, *Chalmers University of Technology and University of Gothenburg*

Bots can Snoop: Uncovering and Mitigating Privacy Risks of Bots in Group Chats **525**
Kai-Hsiang Chou, Yi-Min Lin, Yi-An Wang, and Jonathan Weiping Li, *National Taiwan University;* Tiffany Hyun-Jin Kim, *HRL Laboratories;* Hsu-Chun Hsiao, *National Taiwan University and Academia Sinica*

Double-Edged Shield: On the Fingerprintability of Customized Ad Blockers **529**
Saiid El Hajj Chehade, *EPFL;* Ben Stock, *CISPA Helmholtz Center for Information Security;* Carmela Troncoso, *EPFL and Max-Planck Institute for Security and Privacy* (*MPI-SP*)

Crypto 4: Systems and Protocols

Encrypted Access Logging for Online Accounts: Device Attributions without Device Tracking **533**
Carolina Ortega Pérez and Alaa Daffalla, *Cornell University;* Thomas Ristenpart, *Cornell Tech*

How to Compare Bandwidth Constrained Two-Party Secure Messaging Protocols: A Quest for A More Efficient and Secure Post-Quantum Protocol **535**
Benedikt Auerbach, *PQShield;* Yevgeniy Dodis and Daniel Jost, *New York University;* Shuichi Katsumata, *PQShield and AIST;* Rolfe Schmidt, *Signal Messenger*

S/MINE: Collecting and Analyzing S/MIME Certificates at Scale 539
Gurur Öndarö and Jonas Kaspereit, *Münster University of Applied Sciences;* Samson Umezulike, *Fraunhofer SIT and National Research Center for Applied Cybersecurity ATHENE;* Christoph Saatjohann, *Münster University of Applied Sciences;* Fabian Ising, *Fraunhofer SIT and National Research Center for Applied Cybersecurity ATHENE;* Sebastian Schinzel, *Münster University of Applied Sciences, Fraunhofer SIT and National Research Center for Applied Cybersecurity ATHENE*

Achilles: A Formal Framework of Leaking Secrets from Signature Schemes via Rowhammer 543
Junkai Liang, *Peking University;* Zhi Zhang, *The University of Western Australia;* Xin Zhang and Qingni Shen, *Peking University;* Yansong Gao, *The University of Western Australia;* Xingliang Yuan, *The University of Melbourne;* Haiyang Xue and Pengfei Wu, *Singapore Management University;* Zhonghai Wu, *Peking University*

Comprehensive Deniability Analysis of Signal Handshake Protocols: X3DH, PQXDH to Fully Post-Quantum with Deniable Ring Signatures 545
Shuichi Katsumata, *PQShield & AIST;* Guilhem Niot, *PQShield & Univ Rennes, CNRS, IRISA;* Ida Tucker and Thom Wiggers, *PQShield*

SparSamp: Efficient Provably Secure Steganography Based on Sparse Sampling 549
Yaofei Wang, *Hefei University of Technology;* Gang Pei, *Hefei University Of Technology;* Kejiang Chen and Jinyang Ding, *University of Science and Technology of China;* Chao Pan, Weilong Pang, and Donghui Hu, *Hefei University of Technology;* Weiming Zhang, *University of Science and Technology of China*

A Framework for Designing Provably Secure Steganography 553
Guorui Liao, Jinshuai Yang, Weizhi Shao, and Yongfeng Huang, *Tsinghua University*

Software Security 4: Fuzzing and Other Software Analysis

BLens: Contrastive Captioning of Binary Functions using Ensemble Embedding 557
Tristan Benoit, *Ludwig-Maximilians-Universität München and Bundeswehr University Munich;* Yunru Wang, Moritz Dannehl, and Johannes Kinder, *Ludwig-Maximilians-Universität München and Munich Center for Machine Learning*

TRex: Practical Type Reconstruction for Binary Code 561
Jay Bosamiya, *Microsoft Research;* Maverick Woo and Bryan Parno, *Carnegie Mellon University*

Vest: Verified, Secure, High-Performance Parsing and Serialization for Rust 565
Yi Cai, *University of Maryland, College Park;* Pratap Singh and Zhengyao Lin, *Carnegie Mellon University;* Jay Bosamiya, *Microsoft Research;* Joshua Gancher, *Northeastern University;* Milijana Surbatovich, *University of Maryland, College Park;* Bryan Parno, *Carnegie Mellon University*

TypePulse: Detecting Type Confusion Bugs in Rust Programs 569
Hung-Mao Chen and Xu He, *George Mason University;* Shu Wang, *George Mason University and Palo Alto Networks, Inc.;* Xiaokuan Zhang and Kun Sun, *George Mason University*

From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification 571
Andrew Bao, *University of Minnesota, Twin Cities;* Wenjia Zhao, *Xi'an Jiaotong University;* Yanhao Wang, *Independent Researcher;* Yueqiang Cheng, *MediaTek;* Stephen McCamant and Pen-Chung Yew, *University of Minnesota, Twin Cities*

Network Security 4: Internet and Beyond

A Comprehensive Formal Security Analysis of OPC UA 575
Vincent Diemunsch, *ANSSI and Université de Lorraine, CNRS, Inria, LORIA, France;* Lucca Hirschi and Steve Kremer, *Université de Lorraine, CNRS, Inria, LORIA, France*

Towards Internet-Based State Learning of TLS State Machines 579
Marcel Maehren and Nurullah Erinola, *Ruhr University Bochum;* Robert Merget, *Technology Innovation Institute;* Jörg Schwenk, *Ruhr University Bochum;* Juraj Somorovsky, *Paderborn University*

Haunted by Legacy: Discovering and Exploiting Vulnerable Tunnelling Hosts 583
Angelos Beitis and Mathy Vanhoef, *DistriNet, KU Leuven*

GeCos Replacing Experts: Generalizable and Comprehensible Industrial Intrusion Detection 587
Konrad Wolsing, Eric Wagner, and Luisa Lux, *Fraunhofer FKIE and RWTH Aachen University;* Klaus Wehrle, *RWTH Aachen University;* Martin Henze, *RWTH Aachen University and Fraunhofer FKIE*

Sometimes Simpler is Better: A Comprehensive Analysis of State-of-the-Art Provenance-Based Intrusion Detection Systems 591
Tristan Bilot, *Université Paris-Saclay, LISITE, Isep, and Iriguard;* Baoxiang Jiang, *Xi'an Jiaotong University;* Zefeng Li, *University of British Columbia;* Nour El Madhoun, *LISITE, Isep;* Khaldoun Al Agha, *Université Paris-Saclay;* Anis Zouaoui, *Iriguard;* Thomas Pasquier, *University of British Columbia*

ML and AI Security 4: Robustness

CAMP in the Odyssey: Provably Robust Reinforcement Learning with Certified Radius Maximization 597
Derui Wang, Kristen Moore, Diksha Goel, and Minjune Kim, *CSIRO's Data61 and Cyber Security Cooperative Research Centre;* Gang Li, Yang Li, and Robin Doss, *Deakin University;* Minhui Xue, *CSIRO's Data61 and Cyber Security Cooperative Research Centre;* Bo Li, *University of Chicago;* Seyit Camtepe, *CSIRO's Data61 and Cyber Security Cooperative Research Centre;* Liming Zhu, *CSIRO's Data61*

Robustifying ML-powered Network Classifiers with PANTS 601
Minhao Jin and Maria Apostolaki, *Princeton University*

A Crack in the Bark: Leveraging Public Knowledge to Remove Tree-Ring Watermarks 605
Junhua Lin and Marc Juarez, *University of Edinburgh*

CertTA: Certified Robustness Made Practical for Learning-Based Traffic Analysis 611
Jinzhu Yan, *Tsinghua University;* Zhuotao Liu, *Tsinghua University and Zhongguancun Laboratory;* Yuyang Xie, *Tsinghua University;* Shiyu Liang, *Shanghai Jiao Tong University;* Lin Liu, *National University of Defense Technology;* Ke Xu, *Tsinghua University and Zhongguancun Laboratory*

Invisible but Detected: Physical Adversarial Shadow Attack and Defense on LiDAR Object Detection 615
Ryunosuke Kobayashi, *Waseda University;* Kazuki Nomoto, *Waseda University and Deloitte Tohmatsu Cyber LLC;* Yuna Tanaka and Go Tsuruoka, *Waseda University;* Tatsuya Mori, *Waseda University and NICT and RIKEN AIP*

From Threat to Trust: Exploiting Attention Mechanisms for Attacks and Defenses in Cooperative Perception 617
Chenyi Wang, *University of Arizona;* Raymond Muller and Ruoyu Song, *Purdue University;* Jean-Philippe Monteuuis and Jonathan Petit, *Qualcomm;* Yanmao Man, *Independent Researcher, U.S.;* Ryan Gerdes, *Virginia Tech;* Z. Berkay Celik, *Purdue University;* Ming Li, *University Of Arizona*

System Security 4: Kernel and Low-Level System Security

Await() a Second: Evading Control Flow Integrity by Hijacking C++ Coroutines 621
Marcos Bajo and Christian Rossow, *CISPA Helmholtz Center for Information Security*

System Register Hijacking: Compromising Kernel Integrity By Turning System Registers Against the System . . . 625
Jennifer Miller, Manas Ghandat, Kyle Zeng, Hongkai Chen, Abdelouahab (Habs) Benchikh, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili, *Arizona State University*

When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks 627
Lukas Maar, Lukas Giner, Daniel Gruss, and Stefan Mangard, *Graz University of Technology*

Approximation Enforced Execution of Untrusted Linux Kernel Extensions 631
Hao Sun and Zhendong Su, *ETH Zurich*

EKC: A Portable and Extensible Kernel Compartment for De-Privileging Commodity OS 635
Jiaqin Yan, *Shanghai Jiao Tong University, Southern University of Science and Technology;* Qiujiang Chen, Shuai Zhou, and Yuke Peng, *Southern University of Science and Technology;* Guoxing Chen, *Shanghai Jiao Tong University;* Yinqian Zhang, *Southern University of Science and Technology*

The Cost of Performance: Breaking ThreadX with Kernel Object Masquerading Attacks. 639
Xinhui Shao and Zhen Ling, *Southeast University;* Yue Zhang, *Drexel University;* Huaiyu Yan and Yumeng Wei, *Southeast University;* Lan Luo and Zixia Liu, *Anhui University of Technology;* Junzhou Luo, *Southeast University;* Xinwen Fu, *University of Massachusetts Lowell*

Save what must be saved: Secure context switching with Sailor 641
Neelu S. Kalani, *EPFL and IBM Research - Zurich;* Thomas Bourgeat, *EPFL;* Guerney D. H. Hunt, *IBM T. J. Watson Research Center;* Wojciech Ozga, *IBM Research - Zurich*

Privacy 4: Privacy-Preserving Computation

Flexway O-Sort: Enclave-Friendly and Optimal Oblivious Sorting 645
Tianyao Gu, *Carnegie Mellon University and Oblivious Labs Inc.;* Yilei Wang, *Alibaba Cloud;* Afonso Tinoco, *Carnegie Mellon University and Oblivious Labs Inc.;* Bingnan Chen and Ke Yi, *HKUST;* Elaine Shi, *Carnegie Mellon University and Oblivious Labs Inc.*

Treebeard: A Scalable and Fault Tolerant ORAM Datastore 649
Amin Setayesh, Cheran Mahalingam, Emily Chen, and Sujaya Maiyya, *University of Waterloo*

Learning from Functionality Outputs: Private Join and Compute in the Real World 651
Francesca Falzon, *ETH Zürich;* Tianxin Tang, *Eindhoven University of Technology*

ALERT: Machine Learning-Enhanced Risk Estimation for Databases Supporting Encrypted Queries 655
Longxiang Wang, *City University of Hong Kong;* Lei Xu, *Nanjing University of Science and Technology and City University of Hong Kong;* Yufei Chen, *City University of Hong Kong;* Ying Zou, *Nanjing University of Science and Technology;* Cong Wang, *City University of Hong Kong*

Distributed Private Aggregation in Graph Neural Networks. 659
Huanhuan Jia, Yuanbo Zhao, Kai Dong, Zhen Ling, Ming Yang, and Junzhou Luo, *Southeast University;* Xinwen Fu, *University of Massachusetts Lowell*

Suda: An Efficient and Secure Unbalanced Data Alignment Framework for Vertical Privacy-Preserving Machine Learning. 663
Lushan Song, *Fudan University and ByteDance;* Qizhi Zhang and Yu Lin, *ByteDance;* Haoyu Niu, *Fudan University;* Daode Zhang, *ByteDance;* Zheng Qu and Weili Han, *Fudan University;* Jue Hong, Quanwei Cai, and Ye Wu, *ByteDance*

Assuring Certified Database Utility in Privacy-Preserving Database Fingerprinting 667
Mingyang Song and Zhongyun Hua, *Harbin Institute of Technology, Shenzhen;* Yifeng Zheng, *The Hong Kong Polytechnic University;* Tao Xiang, *Chongqing University;* Guoai Xu, *Harbin Institute of Technology, Shenzhen;* Xingliang Yuan, *The University of Melbourne*

Shechi: A Secure Distributed Computation Compiler Based on Multiparty Homomorphic Encryption 669
Haris Smajlović, *University of Victoria;* David Froelicher, *MIT;* Ariya Shajii, *Exaloop Inc.;* Bonnie Berger, *MIT;* Hyunghoon Cho, *Yale University;* Ibrahim Numanagić, *University of Victoria*

Authentication

Practically Secure Honey Password Vaults: New Design and New Evaluation against Online Guessing 673
Haibo Cheng, Fugeng Huang, and Jiahong Yang, *Peking University;* Wenting Li, *Beijing Institute of Graphic Communication;* Ping Wang, *Peking University*

CertPHash: Towards Certified Perceptual Hashing via Robust Training 677
Yuchen Yang and Qichang Liu, *The Johns Hopkins University;* Christopher Brix, *RWTH Aachen University;* Huan Zhang, *University of Illinois at Urbana–Champaign;* Yinzhi Cao, *The Johns Hopkins University*

Red Bleed: A Pragmatic Near-Infrared Presentation Attack on Facial Biometric Authentication Systems. 681
Bowen Hu, Kuo Wang, and Chip Hong Chang, *Nanyang Technological University*

System Security 5: Securing Systems and Protocols

Oblivious Digital Tokens 685
Mihael Liskij, *ETH Zurich;* Xuhua Ding, *Singapore Management University;* Gene Tsudik, *UC Irvine;* David Basin, *ETH Zurich*

V-ORAM: A Versatile and Adaptive ORAM Framework with Service Transformation for Dynamic Workloads 689
Bo Zhang and Helei Cui, *Northwestern Polytechnical University;* Xingliang Yuan, *The University of Melbourne;* Zhiwen Yu, *Northwestern Polytechnical University and Harbin Engineering University;* Bin Guo, *Northwestern Polytechnical University*

AutoVR: Automated UI Exploration for Detecting Sensitive Data Flow Exposures in Virtual Reality Apps 693
John Y. Kim, Chaoshun Zuo, Yanjie Zhao, and Zhiqiang Lin, *The Ohio State University*

***Found in Translation*: A Generative Language Modeling Approach to Memory Access Pattern Attacks 697**
Grace Jia, Alex Wong, and Anurag Khandelwal, *Yale University*

Current Affairs: A Security Measurement Study of CCS EV Charging Deployments 701
Marcell Szakály, Sebastian Köhler, and Ivan Martinovic, *University of Oxford*

STEK Sharing is Not Caring: Bypassing TLS Authentication in Web Servers using Session Tickets 705
Sven Hebrok, Tim Leonhard Storm, Felix Matthias Cramer, Maximilian Radoy, and Juraj Somorovsky, *Paderborn University*

Vulnerabilities in LLMs: Privacy, Safety, and Defense

Unsafe LLM-Based Search: Quantitative Analysis and Mitigation of Safety Risks in AI Web Search 709
Zeren Luo, Zifan Peng, Yule Liu, and Zhen Sun, *The Hong Kong University of Science and Technology* (*Guangzhou*)*;* Mingchen Li, *The Hong Kong University of Science and Technology* (*Guangzhou*)*; University of North Texas;* Jingyi Zheng and Xinlei He, *The Hong Kong University of Science and Technology* (*Guangzhou*)

Generated Data with Fake Privacy: Hidden Dangers of Fine-tuning Large Language Models on Generated Data . . . 713
Atilla Akkus and Masoud Poorghaffar Aghdam, *Bilkent University;* Mingjie Li, Junjie Chu, Michael Backes, and Yang Zhang, *CISPA Helmholtz Center for Information Security;* Sinem Sav, *Bilkent University*

Cloak, Honey, Trap: Proactive Defenses Against LLM Agents 717
Daniel Ayzenshteyn, Roy Weiss, and Yisroel Mirsky, *Ben Gurion University of the Negev*

Big Help or Big Brother? Auditing Tracking, Profiling, and Personalization in Generative AI Assistants 721
Yash Vekaria, *UC Davis;* Aurelio Loris Canino, *UNIRC;* Jonathan Levitsky, *UC Davis;* Alex Ciechonski, *UCL;* Patricia Callejo, *UC3M;* Anna Maria Mandalari, *UCL;* Zubair Shafiq, *UC Davis*

Private Investigator: Extracting Personally Identifiable Information from Large Language Models Using Optimized Prompts. 725
Seongho Keum and Dongwon Shin, *KAIST;* Leo Marchyok and Sanghyun Hong, *Oregon State University;* Sooel Son, *KAIST*

JBShield: Defending Large Language Models from Jailbreak Attacks through Activated Concept Analysis and Manipulation 729
Shenyi Zhang and Yuchen Zhai, *Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University;* Keyan Guo and Hongxin Hu, *University at Buffalo;* Shengnan Guo, Zheng Fang, and Lingchen Zhao, *Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University;* Chao Shen, *Xi'an Jiaotong University;* Cong Wang, *City University of Hong Kong;* Qian Wang, *Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University*

Web Security

Web Execution Bundles: Reproducible, Accurate, and Archivable Web Measurements 733
Florian Hantke, *CISPA Helmholtz Center for Information Security;* Peter Snyder, *Brave Software;* Hamed Haddadi, *Imperial College London and Brave Software;* Ben Stock, *CISPA Helmholtz Center for Information Security*

The DOMino Effect: Detecting and Exploiting DOM Clobbering Gadgets via Concolic Execution with Symbolic DOM 739
Zhengyu Liu, Theo Lee, Jianjia Yu, Zifeng Kang, and Yinzhi Cao, *Johns Hopkins University*

FIXX: FInding eXploits from eXamples 743
Neil P Thimmaiah, Yashashvi J Dave, Rigel Gjomemo, and V.N. Venkatakrishnan, *University of Illinois Chicago*

Effective Directed Fuzzing with Hierarchical Scheduling for Web Vulnerability Detection 747
Zihan Lin, Yuan Zhang, Jiarun Dai, Xinyou Huang, Bocheng Xiang, Guangliang Yang, Letian Yuan, Lei Zhang, Fengyu Liu, Tian Chen, and Min Yang, *Fudan University*

Crypto 5: HE, MPC, Oblivious Computation

Leuvenshtein: Efficient FHE-based Edit Distance Computation with Single Bootstrap per Cell 751
Wouter Legiest and Jan-Pieter D'Anvers, *COSIC, KU Leuven;* Bojan Spasic and Nam-Luc Tran, *Society for Worldwide Interbank Financial Telecommunication* (*Swift*)*;* Ingrid Verbauwhede, *COSIC, KU Leuven*

Engorgio: An Arbitrary-Precision Unbounded-Size Hybrid Encrypted Database via Quantized Fully Homomorphic Encryption 753
Song Bian, Haowen Pan, Jiaqi Hu, Zhou Zhang, and Yunhao Fu, *Beihang University;* Jiafeng Hua, *Huawei Technology;* Yi Chen and Bo Zhang, *Beijing Academy of Blockchain and Edge Computing;* Yier Jin, *University of Science and Technology of China;* Jin Dong, *Beijing Academy of Blockchain and Edge Computing;* Zhenyu Guan, *Beihang University*

Qelect: Lattice-based Single Secret Leader Election Made Practical 757
Yunhao Wang and Fan Zhang, *Yale University*

H_2O_2RAM: A High-Performance Hierarchical Doubly Oblivious RAM 761
Leqian Zheng, *City University of Hong Kong;* Zheng Zhang, *ByteDance Inc.;* Wentao Dong, *City University of Hong Kong;* Yao Zhang and Ye Wu, *ByteDance Inc.;* Cong Wang, *City University of Hong Kong*

OBLIVIATOR: OBLIVIous Parallel Joins and other OperATORs in Shared Memory Environments. 765
Apostolos Mavrogiannakis, *University of California, Santa Cruz;* Xian Wang, *The Hong Kong University of Science and Technology;* Ioannis Demertzis, *University of California, Santa Cruz;* Dimitrios Papadopoulos, *The Hong Kong University of Science and Technology;* Minos Garofalakis, *ATHENA Research Center and Technical University of Crete*

Efficient Ranking, Order Statistics, and Sorting under CKKS. 769
Federico Mazzone, *University of Twente;* Maarten Everts, *University of Twente and Linksight;* Florian Hahn, *University of Twente;* Andreas Peter, *Carl von Ossietzky Universität Oldenburg*





USENIX Security '25 Artifact Appendix: Easy As Child's Play: An Empirical Study on Age Verification of Adult-Oriented Android Apps

Yifan Yao, Shawn McCollum, Zhibo Sun, Yue Zhang
Drexel University

A Artifact Appendix

A.1 Abstract

GUARD (Guarding Underage Access Restriction Detection) is an automatic tool that analyzes the existence of age verification mechanisms by determining relevant components (e.g., those that can accept the user's age or date of birth) based on the relationships of the components in a layout and tracking the data flows through taint analysis. GUARD contains both a static and a dynamic analyzer. Both analyzers are written in Java; therefore, they are compatible with all operating systems. The static analyzer is based on Soot, while the dynamic analyzer is based on Appium.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

To ensure the compliance of the community standards, it is imperative to strictly adhere to the following guidelines. The guidelines are as follows:

- **Responsible Collection of Data**: A significant number of Android Packages may be required for conducting large-scale research. Therefore, it is essential to ensure that this collection process does not interfere with the website owner's operations. Improperly managed web scraping can disrupt the website's normal functionality. We recommend utilizing AndroZoo; however, we also advise collecting data from additional sources to enhance diversity.
- **Analysis in Controlled Environment**: All analyses must be conducted within a controlled setting, using personal account and machines.

A.2.2 How to access

- Project source code is hosted on Zenodo: https://doi.org/10.5281/zenodo.14676162

A.2.3 Hardware dependencies

- Since GUARD unpacks Android packages into RAM for analysis, it is strongly recommended to run on machines with a minimum of 32GB of RAM.
- To execute the dynamic analyzer, it is required to have an Android emulator or a physical Android device. The emulator can be installed via Android Studio or the Android SDK Command-Line Tools.
- Although GUARD is compatible with all operating systems, it is strongly recommended to run the dynamic analyzer on an emulator (Mac computers with Apple Silicon is recommanded) or phone with an ARM64 architecture.
- At least 100GB of free disk space.

A.2.4 Software dependencies

Although this project is developed in Java, ensuring OS independence, the provided instructions have been written and tested specifically on macOS Sequoia (15.0+).

- Java Development Kit (JDK) 21: https://www.oracle.com/java/technologies/downloads/#java21
- Gradle: https://gradle.org/install/
- Android SDK Command-Line Tools (not necessary to have full Android Studio): https://developer.android.com/studio#command-tools
- Android Build Tools, Android Platform Tools, Android API 33 (Android 13) SDK Platform: `$ sdkmanager "build-tools;33.0.2" "platform-tools" "platforms;android-33"`
- Android System Images (Android 13): `$ sdkmanager "system-images;android-33;google_apis;arm64-v8a"`
- Node.js: https://nodejs.org/en/download/
- Appium: `$ npm i -location=global appium`
- Appium Driver (UiAutomator2): `$ appium driver install uiautomator2`
- (Optional) Docker for build and run container for static analysis.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

```
# clone the repository
git clone https://github.com/Drexel-
    SePAL/AgeScope.git

cd AgeScope

# download sample apks
chmod +x download.sh
./download.sh

# build
gradle build
```

A.3.2 Basic Test

Static Analysier via Docker (results will be placed in `sample/result` folder):

```
# build container image
docker build -t agescope .

# run the container
docker run -it \
    --tmpfs /mnt/ramdisk:rw,size=8g \
    -v $PWD/sample/:/sample agescope

# check results
cat sample/result/*.txt
```

Dynamic Analysier with Android Emulator:

```
# create emulator
avdmanager create avd \
    -n "Pixel_6_0" \
    -d "pixel_6" \
    -k "system-images;android-33;
        google_apis;arm64-v8a"

# start emulator (new terminal window)
emulator -avd Pixel_6_0 \
    -partition-size 8192 \
    -wipe-data

# start Appium server (new terminal
    window)
appium

# run the dynamic analyzer
java -cp ./build/libs/AgeScope-1.0-
    SNAPSHOT.jar DynamicAnalyzer.Main \
     -i sample/apk_index.txt \
     -v 13 \
     -o sample/result \
     -u emulator-5554 # adb devices -l
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): GUARD utilizes a combination of static and dynamic analysis to detect the existence of age verification mechanisms in Android applications.

(C2): Static Analyzer unpacks Android packages and: 1) Extracts the app's XML layout structure to analyze age verification keywords; 2) Tracks input fields that may be used to detect the user's age or identity.

(C3): Dynamic Analyzer loads and processes app layouts in real time by running the app on an emulator or physical device. Once launched, it attempts to interact with the app extensively to trigger and identify the age verification mechanism.

A.4.2 Experiments

General Setup for Experiments:

- **Collect Android Packages**: Please download the Android APK samples we provide by executing `download.sh`. Or you can download the APK samples from the internet. When collecting packages from the internet, besides app identifier and content rating, ensure you gather metadata as much as possible for futher analysis, include but not limited to version number, developer information, and description.

- **Generate Index File for Android Packages**: The index file is a text file that includes the path of apk files.

(E1): *[Static Analysis] [30 human-minutes + 5 compute-minutes + 16GB memory]:*
Preparation: Please refer to **General Setup for Experiments**.
Execution: The static analyzer can be executed via Docker (see **Static Analyzer via Docker**) or natively on macOS by running the `macos_run.sh` script. The script automatically sets up the environment and runs the static analyzer. For more details, refer to `README.md`.
Results: The results of the static analysis are located in the `sample/result` folder, formatted as `<index_file>_result.txt`. The static analyzer outputs the analysis results, including the app's package name and the types of age verification mechanisms found in both layout XML files and activities in details.

(E2): *[Static Analysis] [60 human-minutes + 15 compute-minutes + 16GB memory]:*
Preparation: Please refer to **General Setup for Experiments** and **Dynamic Analysier with Android Emulator**.
Execution: The dynamic analyzer can only run natively and requires an Android emulator or a physical Android device. Before running the analyzer, ensure the emulator or device is connected to the computer, recognized by Android Debug Bridge (adb), and that the Appium server is running. The analyzer will then automatically install the app and perform the analysis.
Results: The static analysis results are stored in the `sample/result` folder, formatted as `<index_file>_exec_result.txt`. The dynamic analyzer outputs the analysis results, including the app's package name and the corresponding string found during execution, which is used to identify age verification mechanisms.

A.5 Notes on Reusability

Due to the varying implementations of age verification across regions, this project can be tailored to identify and adapt to different age verification mechanisms based on their distinct features.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Abusability of Automation Apps in Intimate Partner Violence

Shirley Zhang[†], Paul Chung[‡], Jacob Vervelde[†], Nishant Korapati[†], Rahul Chatterjee[†], Kassem Fawaz[†]

University of Wisconsin–Madison[†]
University of California–San Diego[‡]

A Artifact Appendix

A.1 Abstract

This artifact is used to evaluate iOS Shortcut recipes from four domains: routinehub.co, shareshortcuts.com, shortcutsgallery.com, and matthewcassinelli.com/sirishortcuts. The artifact includes the source code for (1) Shortcut XML parser, (2) code filter, and (3) LLM pipeline to examine the filtered Shortcut recipes. We also include the evaluated datasets and corresponding results.

A.2 Description & Requirements

Our repo structure (link below) is as follows: `deduplicate` folder contains files to calculate the duplicate cases; `llm` folder contains the prompts, code to run LLM pipeline, and the final results; `measurement` folder contains all rationale from LLM, and the results are listed in four folders (each corresponding to a domain: `result_<domain>_test_set`); `parse_xml` and `parse_xml_output` folders contain code files to parse the XML files into YAML format; and `test` folder include all YAML files of Shortcut recipes (stored in folders ending with _yaml), along with copies of these YAML files prepared specifically for LLM evaluation. These copied files meet two criteria: (1) they contain actions corresponding to specific attack operations, and (2) their lengths fit within the LLM's context window.

A.2.1 Security, privacy, and ethical concerns

All codes in the artifacts do not pose privacy and security concerns. However, while the evaluated shortcut recipes are publicly available, the evaluated results should be kept limited to evaluators, as they contain malicious recipes that could pose IPV attacks, and the names of recipes could enable someone to find them online.

A.2.2 How to access

Our artifact is available on the Zenodo platform: https://zenodo.org/records/15522800. The corresponding final DOI is: https://doi.org/10.5281/zenodo.15522800. Please note that the files are limited for public downloading, as the evaluated shortcut recipes could pose IPV attacks. Please contact the authors to download the full artifact.

A.2.3 Hardware dependencies

For system requirements, we use 4 Nvidia A6000 48GB GPUs to run local models, and the environment utilizes VLLM (docs.vllm.ai/en/latest/).

A.2.4 Software dependencies

Our experiments are conducted on Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-60-generic x86_64). The primary tools we used are Python and Jupyter Notebook. We use Conda to manage the environment.

A.2.5 Benchmarks

We collected data from the four domains, routinehub.co, shareshortcuts.com, shortcutsgallery.com, and matthewcassinelli.com/sirishortcuts. We provide only the YAML files, as the data collection time from websites was 2024; therefore, some recipes are no longer available.

A.3 Set-up

We include a requirements.txt that includes all the required packages to run the programs.

A.3.1 Installation

First, create a conda environment `conda create -n test python=3.11 -c conda-forge`, and activate the environment through `conda activate test`, install the environment by running `pip install -r requirements.txt`.

A.3.2 Basic Test

We include a script to check for environment: `check_env.sh`. The environment is successfully set up when this message is printed: `All 234 packages OK.`

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Our pipeline can automatically download an iOS Shortcut referenced by an iCloud link and save it as a local XML file (Section 5.2).*
(C2): *Translate XML representation to YAML text representation (Section 5.2).*
(C3): *Static code analysis to discard shortcuts that are not vulnerable to the four attacks (Section 5.2).*
(C4): *Use Qwen2.5-32B-Instruct model to produce both a rationale and a trinary (yes/no/maybe) prediction (Section 5.2).*

A.4.2 Experiments

Please also refer to README.md.

(E1): iCloud link → XML, 5 human-minutes
- **Preparation:** Obtain an iCloud Shortcut link (either from your own device or any public link) and paste in the `run(<link>)` in `get_xml_file.py`.
- **Execution:** `$ python get_xml_file.py`
- **Results:** A file named `<shortcut>.xml` appears in the working directory, confirming (C1).

(E2): XML → YAML, 5 human-minutes, machine-minutes are dependent on input size
- **Preparation:** Place one or more `*.xml` files produced by (E1) in `/test/` (The provided dataset does not contain XML files).
- **Execution:** `$ python test/test.py <input_file_or_directory> -o <output_path>`
- **Results:** This produces YAML under `/test/test_..._result_yaml`. Confirming (C2).

(E3): Static Code Analysis & Category Calculation, 5 human-minutes, machine-minutes are dependent on input size
- **Preparation:** Ensure the YAML files from (E2) are present in: `/test/test_[domain_name]_result_yaml`. (We provided these files as part of our results.)

Execution: 1. Run this script:
`$ python measurement/static_code_analysis.py`
(by default, the input folder is:
`/test/test_[domain_name]_result_yaml`)
2. Run this script:
`$ python measurement/calculate_result_of_category.py`
(by default, the input folder is:
`/test/test_result_code_analysis_[domain_name]`
Results: This is the code filter part we mentioned in the paper (Section 5.2), and it writes output files to
`/test/test_result_code_analysis_[domain_name]`
After calculating the result of each category, it will generate text files containing the filenames for each of the four attacks. By default, the final output folder for this step is
`/test/output_results_[domain_name]`
Confirms (C3): only shortcuts potentially vulnerable to the four attacks proceed to the LLM stage.

(E4): LLM Rationale + Trinary Verdict, 30 human-minutes + model-dependent compute-time (by default, we use Qwen2.5 32B-Instruct through huggingface and vllm, the processing time is more than 24 hours for all four datasets with 4 Nvidia A6000 GPUs)
Preparation: Pick an LLM (the default in our paper is **Qwen2.5-32B-Instruct**) and, if needed, adjust `max_tokens` in `/llm/distributor.py` to match your compute budget.
Execution: 1.Run this script:
`$ python llm/distributor.py`
This script creates one subfolder per attack under `/test/output_results_[domain_name]/action_categories`.
Example usage:

```
copy_matching_files("[attack]_to_evaluate_[domain]",
"test_[domain_name]_result_yaml",
"output_results_[domain]/action_categories/[attack].txt_safe.txt")
```

2.Run this script
`$ python test/move_safe_files.py` to copy only the files safe for a given attack/domain
3. Launch `llm/run_llm.ipynb` *or* `llm/run_gpt.py` (used for the impersonation re-run) to generate full rationales.
Please modify the file path in the code to match your local environment.
Additionally, you will need to have an OpenAI API to complete this step.
Our default input path is
`../test/[attack_category]to_evaluate[domain_name]`
, and default output folder is
`../measurement/result_[domain_name]_test_set`
. Note: since for impersonation, Qwen gives a high false positive rate, we add a second step that uses gpt-4o in /llm/run_gpt.py.

4. Run `llm/run_binary_answer.ipynb` to extract the final `yes/no/maybe` decisions. For files processed by GPT, they have a different output format, so pick the corresponding function from (extract_last_prompt_json_gpt, extract_last_prompt_json), extract_last_prompt_json is for Qwen.
Results: This step writes CSVs to `/llm/final_result_csvs` containing
(i) the LLM rationale and (ii) the trinary verdict for every processed shortcut, thereby confirming (C4).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Voluntary Investments, Mandatory Minimums, or Cyber Insurance: What Minimizes Losses?

Adam Hastings

A Artifact Appendix

A.1 Abstract

These artifacts contain the following:

- The code needed to perform the empirical work in our paper
- The datasets needed to reproduce the figures in the paper
- The code (and runtime environment) needed to run the simulations and reproduce the data used in the paper

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We do not forsee any security, privacy, or ethical concerns with running this artifact.

A.2.2 How to access

The codebase for this work is maintained at http://github.com/columbiacastl/monte-carlo-security-games/. We also provide a stable reference via Zenodo at https://doi.org/10.5281/zenodo.14728685.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

We have tried to make our artifacts as portable as possible. To this end we recommend installing the following:

1. Docker: https://docs.docker.com/engine/install/
2. Some Linux command line tools: `tar`, `curl`, `git`

We recommend using a Linux environment for artifact evaluation.

A.2.5 Benchmarks

None.

A.3 Set-up

We provide two methods of installation. The first—Zenodo—contains a stable artifact *and all data used in the paper*. This is necessary for a strict verification of Experiment E2 below. You may also install via GitHub which contains code but not data. Via the GitHub installation, you may *reproduce* the data used in the paper (Experiment E3) and then verify that the data matches the data in the paper (Experiment E2).

A.3.1 Installation via Zenodo

```
# download the containerized application
wget https://zenodo.org/records/15042431/files/
    artifacts.tar.gz

# extract and change directories
tar -xzvf artifacts.tar.gz
cd monte-carlo-security-games/
```

A.3.2 Installation via GitHub

```
git clone https://github.com/columbia-castl/monte-
    carlo-security-games.git
cd monte-carlo-security-games/
```

A.3.3 Building the Docker container

If you have a Docker environment, you can build a Docker image from the Dockerfile provided in the repository. This step may take 10 minutes to complete.

```
# pull a stable version of ubuntu
sudo docker pull ubuntu:22.04

# build the docker image from the Dockerfile
sudo docker build -t ae_image .

# run the docker image in an interactive container
sudo docker run -it ae_image
```

A.3.4 Basic Test

Once you have a shell to the running docker container, execute the following within the container:

```
cd root/simulator/

# build the simulator
make release

# run a small input
./run/release/run_games configs/fullsize_tiny.json
```

You may be prompted to overwrite the existing logfile for the input config `fullsize_tiny.json`. If so type "y" to proceed. The output should look something like this:

```
Creating logs/fullsize_tiny.csv

This file already exists: logs/fullsize_tiny.csv
Do you want to replace it (Y)? Or append to it (A)? Y/A/n
 >> y
started 1 games at Tue Feb 25 22:27:50 2025
finished computation at Tue Feb 25 22:28:02 2025
elapsed time: 11.2933s
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1) The curve fittings and regressions in Section 3 of the paper follow from the data acquired.
(C2) The model output data generates the figures in Sections 5–10 of the paper.
(C3) The model produces data similar to the data used in (C2).

A.4.2 Experiments

All experimental verification will happen inside the running docker container.

(E1): *Verifying (C1) [10 human-minutes + 1 compute-minute]: ...*

```
cd /root/parameter-calcs/scripts

# Re-create wealth regression (Section 3.2)
# parameters will be included in the output
# mu=1.1356, sigma=-1.1184
# other outputs located in ../figures/wealthfitting
python3 fit_marketcap_revenue_curves.py

# Re-create Figure 1 (Section 3.3)
# output will be in ../figures/ransom_regression/
python3 ransom_regression_plot.py

# Re-create security posture regression
# output will be in ../figures/posture_fitting
python3 curve_fit_posture.py

# Re-create Figure 2 (Secton 3.6)
# output will be in /root/parameter-calcs/figures/erf/
python3 plot_erf.py
```

(E2): *Verifying (C2) [10 human-minutes + 10 compute-minute]: ...*

```
cd /root/simulator/scripts/

# Generate baseline model figures (Section 5)
# outputs (Figures 3--6)  located figures/fullsize_short/
python3 run_all.py ../logs/fullsize_short.csv

# Generate the sensitivity analysis
# output  (Figure 7) located in figures/sensitivity_analysis/
sensitivity_analysis_MAX_ITERATIONS=500*
python3 plot_sensitivity_analysis.py

# Generate figures for the mandated securty investments model
# outputs (Figure 8--10) in figures/fullsize_short_MANDATORY_INVESTMENT*/
python3 ./run_all.py ../logs/fullsize_short_MANDATORY_INVESTMENT\=0.01.csv
python3 ./run_all.py ../logs/fullsize_short_MANDATORY_INVESTMENT\=0.02.csv
python3 ./run_all.py ../logs/fullsize_short_MANDATORY_INVESTMENT\=0.03.csv
python3 ./run_all.py ../logs/fullsize_short_MANDATORY_INVESTMENT\=0.04.csv
python3 ./run_all.py ../logs/fullsize_short_MANDATORY_INVESTMENT\=0.05.csv


# Generate figures for the mandatory insurance model
# outputs (Figure 11--12) in figures/fullsize_short_mandatory_insurance
python3 ./run_all.py ../logs/fullsize_short_mandatory_insurance.csv

# Generate figures for the actuarially fair model
# outputs (Figures 13--14) in figures/fullsize_short_selfless_insurers
run_all.py ../logs/fullsize_short_selfless_insurers.csv

# Generate figures for model with growth
# outputs (Figures 15--16) in figures/fullsize_short_with_asset_growth_GROWTH_RATE=*/
python3 run_all.py ../logs/fullsize_short_with_asset_growth_GROWTH_RATE\=0.001.csv
python3 run_all.py ../logs/fullsize_short_with_asset_growth_GROWTH_RATE\=0.002.csv
python3 run_all.py ../logs/fullsize_short_with_asset_growth_GROWTH_RATE\=0.01.csv
python3 run_all.py ../logs/fullsize_short_with_asset_growth_GROWTH_RATE\=0.02.csv
python3 run_all.py ../logs/
```

```
    fullsize_short_with_asset_growth_GROWTH_RATE
    \=0.04.csv
```

(E3): *Verifying (C3) [1 human-hour + 20 compute-hours]:* ...
To reproduce the data used in (C2), you can re-run the simulator on all inputs.

```
cd /root/simulator/
make release # build the release version if
    you  haven't already

# Run the baseline model
./run/release/run_games configs/fullsize_short
    .json

# Run the parameter sweeps for sensitivty
    analysis
# NOTE: the sweep simulations require the
    debug binary for serial execution!
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/
    sweep_ATTACKS_PER_EPOCH.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/
    sweep_CTA_SCALING_FACTOR.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/sweep_DEPRECIATION.
    json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/sweep_INEQUALITY.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/
    sweep_INVESTMENT_SCALING_FACTOR.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/sweep_LOSS_RATIO.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/sweep_NUM_QUOTES.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/sweep_RANSOM_B0.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/sweep_RANSOM_B1.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/
    sweep_RECOVERY_COST_BASE.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/
    sweep_RECOVERY_COST_EXP.json
./run/debug/run_games configs/sweeps/
    MAX_ITERATIONS=500/
    sweep_RETENTION_REGRESSION_FACTOR.json


# Run the mandatory investment simulations
./run/release/run_games configs/
    fullsize_short_MANDATORY_INVESTMENT=0.01.
    json
./run/release/run_games configs/
    fullsize_short_MANDATORY_INVESTMENT=0.02.
    json
./run/release/run_games configs/
    fullsize_short_MANDATORY_INVESTMENT=0.03.
    json
./run/release/run_games configs/
    fullsize_short_MANDATORY_INVESTMENT=0.04.
    json
./run/release/run_games configs/
    fullsize_short_MANDATORY_INVESTMENT=0.05.
    json


# Run the mandatory insurance simulations
./run/release/run_games configs/
    fullsize_short_mandatory_insurance.json

# Run the actuarially fair simulations
./run/release/run_games configs/
    fullsize_short_selfless_insurers.json

# Run the simluations with growth
./run/release/run_games configs/
    fullsize_short_with_asset_growth_GROWTH_RATE
    =0.001.json
./run/release/run_games configs/
    fullsize_short_with_asset_growth_GROWTH_RATE
    =0.002.json
./run/release/run_games configs/
    fullsize_short_with_asset_growth_GROWTH_RATE
    =0.01.json
./run/release/run_games configs/
    fullsize_short_with_asset_growth_GROWTH_RATE
    =0.02.json
./run/release/run_games configs/
    fullsize_short_with_asset_growth_GROWTH_RATE
    =0.04.json
```

A.5 Notes on Reusability

In addition to the above arfact assets, we also include a live online demo of our model. This allows for users to run the model on custom inputs via a web browser and is much more accessible than the containerized version above. However, for usability reasons it only runs one instance of the game at a time and so it would be impractical to use it to fully reproduce the work above. However, it provides a low-resolution approximation with may be useful regardless. It is available here: https://cyberspending.cs.columbia.edu/

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: A First Look at Governments' Enterprise Security Guidance

Kimberly Ruth
Stanford University
kcruth@cs.stanford.edu

Raymond Buernor Obu
Stanford University
oburay@stanford.edu

Ifeoluwa Shode*
Fisk University
josephshode@gmail.com

Gavin Li
Stanford University
hawklin@cs.stanford.edu

Carrie Gates
FS-ISAC
cgates@fsisac.com

Grant Ho
University of Chicago
grantho@uchicago.edu

Zakir Durumeric
Stanford University
zakir@cs.stanford.edu

A Artifact Appendix

A.1 Abstract

Governments globally publish a wealth of cybersecurity guidance for enterprises; however, little prior work has studied this guidance ecosystem. This paper presents the first systematic investigation of governments' enterprise security guidance. We build a corpus of prominent guidance documents from 41 countries, develop a tree-based content taxonomy and quantitative comparison metric, and compare a selection of ten guidance documents' content. Our analysis shows that governments differ widely in what and how they advise companies.

A.2 Description & Requirements

Our artifact materials contain the following main components:

- `Guidance document links.xlsx`: Our dataset of links to countries' guidance resources, sorted by type. Functionally, this spreadsheet is a qualitative codebook constructed during our document search process as described in our paper (Section 3.1.2). For each resource we found during our search, we determined its category according to the high-level document typology we outline in our paper (Section 3.2), and added its URL directly to the corresponding spreadsheet column. We then added columns to provide a more granular breakdown of sector-, tech-, threat-, and mitigation-specific documents (using our qualitative observations of common elements within each of those four categories).
- `Roots of shrubs.xlsx` and `v2 Shrubs to trees.xlsx`: Our hierarchical taxonomy and our mappings of ten guidance documents. The former spreadsheet contains the first three levels of our taxonomy; the latter spreadsheet contains the final two levels.
- `tree_similarity.ipynb`: Our code for our custom tree edit distance metric. Usage and supplementary files are described in `README.md`.

In addition, our `v2 Shrubs to trees.xlsx` document contains supplementary data in the tabs `themes_count`, `subthemes_count`, and `controls_count`. These are simple raw counts over the data structure that `tree_similarity.ipynb` constructs (with the addition of "Other" subthemes and controls). The data structure records, for each country document, which nodes in our taxonomic tree the document contains content for. The supplementary data counts how many country documents contain content for each node in the taxonomic tree.

A.2.1 Security, privacy, and ethical concerns

Our artifact materials involve no security, privacy, or ethical concerns for evaluators. This work only involves analysis of publicly available data, without sensitive or identifying information about individuals or groups. Our code is non-destructive and can be run safely on evaluators' machines.

A.2.2 How to access

All artifact materials are accessible through Zenodo at the following URL: https://doi.org/10.5281/zenodo.15612457

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Our code requires `pygraphviz` as well as a suite of supplementary Python packages viewable in the `requirements.txt` file. Full installation instructions, using `uv` for dependency managemment, are provided in the repository's README.

*Work done while visiting Stanford University.

A.2.5 Benchmarks

Our tree similarity code requires two specially-formatted input data files: one capturing the first three levels of our content taxonomy, the other capturing the remaining two levels. Both data files are provided in our artifact materials, as `Roots of shrubs.xlsx` and `v2 Shrubs to trees.xlsx` respectively.

A.3 Set-up

A.3.1 Installation

Download all files from the artifact repository and place them together into the same directory. To install dependencies for our tree similarity code:

1. Install `pygraphviz`. On Ubuntu LTS:
   ```
   sudo apt-get install graphviz graphviz-dev
   pip install pygraphviz
   ```
   Non-Ubuntu installation instructions and troubleshooting tips are linked from the repository's README.

2. We suggest `uv` for dependency management. To install `uv`:
   ```
   curl -LsSf https://astral.sh/uv/install.sh
   | sh
   ```

3. Install the remaining Python dependencies listed in `requirements.txt`. Using `uv`:
   ```
   uv sync
   ```
   Or, if running from within a Python `venv`:
   ```
   uv sync --active
   ```

A.3.2 Basic Test

For our dataset of government guidance document links, open and view `Guidance document links.xlsx`. The file should contain one row of data for each of the 41 countries we included in our search.

For our tree similarity code, create an `out/` directory in the same location as the code, then open and run `tree_similarity.ipynb` in Jupyter Lab. Key functions take the two input file names (defaulting to `Roots of shrubs.xlsx` and `v2 Shrubs to trees.xlsx`) as parameters. The code should generate the following output files in the `out/` directory:

1. `pairwise_tree_differences.csv`: Raw (non-normalized) tree distances between pairs of documents.

2. `doc_tree_to_full_tree_cost.csv`: Raw (non-normalized) tree distance from each individual document's tree to the total taxonomic tree, used for computing coverage numbers.

3. `tree_*.png` and `full_tree.png`: Visualizations of each document's content tree as well as of the total taxonomic tree.

4. `venn_us_au_uk.png`: Venn diagram showing overlap in controls from three key allied countries.

A.4 Evaluation workflow

Below we outline the major claims of our work (§A.4.1) and describe the steps to validate these major claims (§A.4.2).

A.4.1 Major Claims

(C1): Governments commonly present guidance in a wide array of differently scoped documents. This is illustrated in Table 1 (Section 3.2) and supported by experiment (E1).

(C2): Governments vary widely in the breadth and amount of implementation detail they consider "essential" guidance for companies. This is illustrated in Figure 2 (Section 4.3) and supported by experiment (E2).

(C3): There is little consensus among governments—even between close allies—about what controls are most essential for companies to implement. This is illustrated in Figures 3, 4, 5, and 6 (Section 4.4), and supported by experiments (E3), (E4), and (E5).

(C4): Governments present implementation details for key controls inconsistently, even exhibiting direct contradictions. This is described in Section 4.5 and supported by experiment (E6).

A.4.2 Experiments

(E1): [2 human-hours] Verify guidance availability.
Preparation: Open and inspect `Guidance document links.xlsx`.
Execution: Inspect the data for the following ten document categories (spreadsheet column labels parenthesized): General Purpose (F), Timely (AI), Size-Specific (AJ), Critical Infra (AU), Sector Specific - Any (AK), Tech Specific - Any (AV), Threat Model Specific - Any (H), Mitigation Specific - Any (S), For Individuals (BJ), Professional Development (BK).
Results: Compare nonempty spreadsheet cells with filled circles in Table 1 to validate table data. Observe that a majority of rows in the table have guidance available (a filled circle or a number greater than zero) for at least six of these ten guidance document categories.

(E2): [20 human-minutes + 5 compute-minutes] Validate content trees.
Preparation: Follow installation instructions in §A.3.1.
Execution: Run `tree_similarity.ipynb` as described in §A.3.2.

Results: Inspect the files titled `tree_*.png` in the `out/` directory. Note that each file name contains an identifying two-letter country code. Compare with Figure 2.

(E3): [20 human-minutes] Verify pairwise country differences.
Preparation: Open `pairwise_tree_differences.csv` from the `out/` directory generated from running `tree_similarity.ipynb` in experiment (E2). Open `v2 Shrubs to trees.xlsx` and go to the `pairwise_tree_differences` tab.
Execution: Verify that the data in `pairwise_tree_differences.csv` matches that in the `pairwise_tree_differences` tab. Note that rows may not appear in the same order.
Results: Compare the "Normalized" column numbers in the `pairwise_tree_differences` tab with the corresponding numbers in Figure 3. Observe that most pairs of country documents differ by over 50%.

(E4): [10 human-minutes] Verify distributions of content coverage.
Preparation: Open `v2 Shrubs to trees.xlsx`.
Execution: Inspect the tabs labeled `themes_count`, `subthemes_count`, and `controls_count`.
Results: Compare `themes_count` with Figure 4, and `subthemes_count` and `controls_count` with Figure 5. Observe that only a minority of content attains widespread consensus to include.

(E5): [5 human-minutes] Validate allied-countries results.
Preparation/Execution: Open `venn_us_au_uk.png` from the `out/` directory generated from running `tree_similarity.ipynb` in experiment (E2).
Results: Compare with Figure 6.

(E6): [30 human-minutes] Validate variance in implementation details.
Preparation: Open `v2 Shrubs to trees.xlsx`.
Execution: Inspect the tabs labeled with control names, especially `Backup Operation`, `Vulnerability Remediation`, and `Password Authentication`.
Results: Observe the data sparsity in the tables, where each attribute value typically appears in a minority of the guidance documents. Note lines 42-43 in the `Password Authentication` tab, which is an example of contradictory advice.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Characterizing and Detecting Propaganda-Spreading Accounts on Telegram

Klim Kireev[§†], Yevhen Mykhno, Carmela Troncoso[§†], Rebekah Overdorf[‡⋆]

§ EPFL, † MPI-SP Max Plank Institute for Security and Privacy

‡ Ruhr University Bochum (RUB), Research Center Trustworthy Data Science and Security in University Alliance Ruhr

⋆ University of Lausanne

A Artifact Appendix

A.1 Abstract

The present submission contains the dataset used in the paper, embeddings computed on this dataset, model checkpoints required for the functionality analysis, and source code both for the inference of the attached models and for training new ones if such training is useful for future research. In addition to that, we also provide the source code in the form of the Jupiter notebook, which can be used to build most figures present in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We do not see any risk for evaluators coming from this artifact evaluation procedure.

A.2.2 How to access

Our artefact can be downloaded from Zenodo *https://zenodo.org/records/14736756*

A.2.3 Hardware dependencies

Most of the code needed to run the machine learning models requires the CUDA-enabled GPU. In particular, to train the attached models, we used NVIDIA RTX 4070. However, less powerful NVIDIA GPUs (any NVIDIA GPU with CUDA support and >2GB of VRAM) can also be used in this evaluation. Besides that, we recommend using a machine with >=32GB of RAM available in order to minimize page swapping. Finally, approx. 15GB of free disk space is required.

A.2.4 Software dependencies

The evaluation requires GNU/Linux operating system (we used Ubuntu 22.04), with CUDA 12.2 installed (our nvidia driver version: 535.183.01). The installation of the Python environment used in the report is done via the conda package manager (we used Miniconda 4.13.0).

A.2.5 Benchmarks

The dataset used in the evaluation is part of the artifact.

A.3 Set-up

A.3.1 Installation

First, evaluators need to download the artifact from Zenodo and unzip all of them in a single folder (we recommend creating a separate folder); source code files should be in the same folder as the dataset CSV file.

After this step, the main Python dependencies must be installed via commands listed in the INSTRUCTIONS.md file (conda ... , pip ...).

A.3.2 Basic Test

The functionality test can be run via: `python model_eval.py` and the output should be:

```
The dataset is loaded
Overall Accuracy: 83.5
New Topics Accuracy: 86.842
Validation Accuracy: 74.0
```

The script execution time can be 1-10 min.

A.4 Evaluation workflow

A.4.1 Major Claims

Major claims made in the paper:

(C1): Propaganda accounts and user accounts demonstrate different properties in terms of message length, lifespan, number of active channels, and number of messages (Section 3.3, Figures 4-7). However, regarding the number of replies per message sent, user and propaganda accounts are similar (Figure 8), demonstrating that propaganda

accounts are efficient in invoking users' reactions in the form of replies. These properties are demonstrated in the experiment E1.

(C2): The Trigger-Propaganda embeddings detector outperforms other detection methods present in the paper with respect to performance on the new topics, clean accuracy, and performance on the validation propaganda network (Table 3, Section 4.3). It is proven by the experiment E2.

A.4.2 Experiments

(E1): *[Data Analysis] [30-60 human-minutes + 1 compute-hour + 10GB disk + 20-32 GB RAM]*: The aim of the data analysis is to generate the figures present in the paper.

Preparation: First, the jupyter notebook server must be launched via command: `jupyter notebook`, then select the file `figures.ipynb` in the opened window (click the URL in the output if the window in the browser does not open)

Execution: Run all the cells in the notebook using the button "Run all cells"

Results: Resulting figures should match figures present in the paper

(E2): *[Detection performance] [10 human-minutes + 0.5-1 compute-hour]*: This experiment shows that the provided models function and produce the results reported in Table 3.

Preparation: Open the terminal in the folder with the dataset and the source code.

Execution: Run `python models_eval.py`, followed by `python handcrafted_eval.py`

Results: Resulting numbers should match the numbers reported in Table 3.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: GradEscape: A Gradient-Based Evader Against AI-Generated Text Detectors

Wenlong Meng[†] Shuguo Fan[†] Chengkun Wei[†] Min Chen[≀] Yuwei Li[♮§]
Yuanchao Zhang[‡] Zhikun Zhang[†] Wenzhi Chen[†]
[†]Zhejiang University [≀]Vrije Universiteit Amsterdam [♮]National University of Defense Technology
[§]Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation [‡]Mybank, Ant Group

A Artifact Appendix

A.1 Abstract

We proposed GradEscape, the first gradient-based evader for attacking AI-generated text (AIGT) detectors. GradEscape overcomes the undifferentiable computation problem, caused by the discrete nature of text, by constructing weighted embeddings for the detector input. It then updates the evader model parameters using feedback from victim detectors, achieving high evasion rates with minimal text modifications.

This artifact is used to evaluate GradEscape against a range of detectors. It includes code for reproducing GradEscape as well as three baseline evaders. We provide a Python library that makes it easy to reproduce existing detectors and evaders. The artifact also contains trained evader models and scripts for replicating our experiments on two real-world AIGT detectors, namely Sapling and Scribbr. To help mitigate the risks posed by AIGT evaders, we showcase that our proposed active paraphrase defense can effectively reduce evasion rates.

A.2 Description & Requirements

We package the artifact into two separate files: `GradEscape.zip` and `Usenix-AE.zip`. `GradEscape.zip` contains the source code; while `Usenix-AE.zip` contains evaluation datasets and trained models. This section describes the minimal hardware and software requirements needed to run the artifact.

A.2.1 Security, privacy, and ethical concerns

Our attacks target exclusively at the target AIGT detector. Therefore, there is no security risk for evaluators. All benchmark datasets are sourced from open corpora, so there are no privacy concerns. Regarding real-world experiments, Sapling and Scribbr have updated their services with stronger models. Therefore, our artifact poses no significant risk to these two online platforms.

A.2.2 How to access

Our artifact is available through Zenodo. The artifact can be accessed at https://doi.org/10.5281/zenodo.15586856.

A.2.3 Hardware dependencies

Two Nvidia RTX A6000 GPUs are the minimum GPU requirement to run the artifact. We recommend at least a 20-core CPU, 32 GB RAM, and 256 GB of free disk space.

A.2.4 Software dependencies

- **OS:** Ubuntu 20.04+. A macOS machine is needed to open Scribbr webarchive file. If you encounter a security warning when opening the webarchive file, please go to Settings, Privacy & Security and click open anyway.
- **Package Manager:** Conda.
- **API Key:** A Sapling API key is required to run Sapling experiments. Conducting these experiments entails money costs.

A.2.5 Benchmarks

Our evaluation requires four datasets: GROVER, HC3, GPA, and GPTWiki. GROVER and GPA are provided in `Usenix-AE.zip`. Our code will automatically download and manage HC3 and GPTWiki. Some pre-trained victim detectors are also included in `Usenix-AE.zip`.

A.3 Set-up

The setup utilizes Conda for environment management.

A.3.1 Installation

Install Main Environment. Download `GradEscape.zip` and `Usenix-AE.zip`. Unzip and place them in the same directory. Then, use the following commands to create an environment:

```
conda create -n ge python=3.10
conda activate ge
cd GradEscape
./install.sh
cp src/AIGT/.config.yaml src/AIGT/config.yaml
```

Generate Word Similarity Matrix. Perturbation-based evaders rely on a word similarity matrix to select synonyms.

```
cd Usenix-AE
git clone
↪ https://github.com/nmrksic/counter-fitting.git
cd counter-fitting/word_vectors/
unzip counter-fitted-vectors.txt.zip
python ../../../GradEscape/tools/comp_cos_sim_mat.py
↪ counter-fitted-vectors.txt
```

Then edit `config.yaml` to set the correct data_dir and counter_fitting_path.

Create vLLM Environment. We need vLLM for fast paraphrasing. Since vLLM has complex dependencies, we create a new environment specifically for vLLM.

```
conda create -n vllm python=3.10
conda activate vllm
cd GradEscape
pip install -r paraphrase_requirements.txt
```

Our artifact mainly runs on `ge`; `vllm` is only used for paraphrase defense.

A.3.2 Basic Test

We provide a simple test that trains a detector using `AIGT`.

```
python basic_test.py
```

If you see "Test finished successfully!", it means the installation was successful.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): GradEscape achieves higher evasion rates than state-of-the-art evaders under the same text quality requirement. This is proven by the experiment (E1) described in Section 6.2 whose results are illustrated in Figure 3 and Figure 4.

(C2): GradEscape is effective against real-world commercial AIGT detectors. This is proven by experiment (E2) described in Section 6.6 whose results are reported in Table 4 and illustrated in Figure 20 and Figure 21.

(C3): Our proposed potential defense can reduce evasion rates below 0.2 against multiple evaders. This is proven by experiment (E3) described in Section 7 whose results are illustrated in Figure 11.

A.4.2 Experiments

(E1): *[Verify Evasion Effectiveness] [5 human-minutes + 1 compute-hour]:*

How to: Navigate to `examples` directory; activate `ge` environment; execute evader training script in `scripts`. The evasion rate and text quality metrics will be printed on the shell.

Preparation: None.

Execution: Navigate to `examples` directory and activate `ge`:

```
cd examples
conda activate ge
```

Then execute the evader training script:

```
./scripts/train_evader_roberta_grover.sh
```

Results: After the program finishes running, the evasion rate and text quality metrics (perplexity, cosine similarity, GRUEN, and ROUGE) will be printed to the terminal. Evaluators can compare these results with the first row of Figure 3. For ease of reference, we provide details of text quality metrics in Table 1.

Table 1: Text quality metrics summary. Expected refers to typical value ranges on GROVER; Print % indicates whether the metric is displayed as a percentage.

Metric	Range	Expected	Larger Better	Print %
Perplexity	$[1, +\infty)$	$[10, 40]$	No	No
Cos-sim	$[-1, 1]$	$[0.95, 1.00]$	Yes	No
GRUEN	$[0, 1]$	$[0.6, 0.8]$	Yes	No
ROUGE	$[0, 1]$	$[0.8, 1.0]$	Yes	Yes

(E2): *[Real-world Case Studies] [5 human-minutes + 30 compute-minutes]:*

How to: Set your Sapling API key environment parameter; run the two real-world case study Jupyter Notebooks. Evaluators may open `Scribbr.webarchive` to verify Scribbr results.

Preparation: A Sapling API key and a macOS machine for Scribbr verification.

Execution: Set your Sapling API key:

```
export SAPLING_API_KEY=<your_api_key>
```

Run `real_world_demo_sapling.ipynb` and `real_world_demo_scribbr.ipynb`. The execution environment is `ge`.

Results: The Sapling results will be printed in its Jupyter Notebook. Evaluators can compare the printed results with Figure 20 and Table 4. Verifying Scribbr results requires copying the output into the website rendered by `Scribbr.webarchive`. The Scribbr results should be the same as Figure 21.

(E3): *[Paraphrase Defense Experiment] [5 human-minutes + 2 compute-hours]:*
How to: Navigate to `examples` directory. First, use `vllm` environment to run `paraphrase_defense_grover.sh`. Then, activate `ge` and run `eval_paraphrase_defense_grover.sh`. It will generate a figure in the current directory named `paraphrase_defense_grover.pdf`.
Preparation: None.
Execution: Navigate to `examples`:

```
cd examples
```

Run paraphrase:

```
conda activate vllm
./scripts/paraphrase_defense_grover.sh
```

Train a new detector and evaluate the defense:

```
conda activate ge
./scripts/eval_paraphrase_defense_grover.sh
```

Results: The program will generate a figure named `paraphrase_defense_grover.pdf` in `examples/`. This figure illustrates evasion rates before and after applying our defense. Evaluators can compare the generated figure with Figure 11 in our paper.

A.5 Notes on Reusability

We implement `AIGT` in a reusable way. All configurations, including datasets, detectors, and evaders, are extracted in `arguments.py`. Followers can replicate existing AIGT detectors and evaders with little effort. The user interface of detectors and evaders is designed in a unified and HuggingFace-like way. Followers can also write their own detectors and evaders in `detectors/` and `evaders/`, respectively.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: HATEBENCH: Benchmarking Hate Speech Detectors on LLM-Generated Content and Hate Campaigns

Xinyue Shen[1] Yixin Wu[1] Yiting Qu[1] Michael Backes[1] Savvas Zannettou[2] Yang Zhang[1]
[1]*CISPA Helmholtz Center for Information Security* [2]*Delft University of Technology*

A Artifact Appendix

A.1 Abstract

In this paper, we propose HateBench, a framework for benchmarking hate speech detectors on LLM-generated hate speech. We first construct a hate speech dataset of 7,838 samples generated by six widely-used LLMs covering 34 identity groups, with meticulous annotations by human labelers. We then assess the effectiveness of eight representative hate speech detectors on the LLM-generated dataset. Our results show that while detectors are generally effective in identifying LLM-generated hate speech, their performance degrades with newer versions of LLMs.

We also reveal the potential of LLM-driven hate campaigns, a new threat that LLMs bring to the field of hate speech detection. By leveraging advanced techniques like adversarial attacks and model stealing attacks, the adversary can intentionally evade the detector and automate hate campaigns online. The most potent adversarial attack achieves an attack success rate of 0.966, and its attack efficiency can be further improved by 13-21x through model stealing attacks with acceptable attack performance.

A.2 Description & Requirements

This artifact includes code to generate the main result tables in the paper, including:

- Table 3: Performance on LLM-generated samples.
- Table 4: F1-score on LLM-generated and human-written samples.
- Table 6: Performance of adversarial hate campaign
- Table 8: Performance of model stealing attacks.
- Table 9: Performance of stealthy hate campaign with black-box attacks.
- Table 10: Performance of stealthy hate campaign with white-box gradient optimization.

Tables 3 and 4 can be executed directly on a local PC without requiring a GPU environment. For other experiments, we recommend using an environment with NVIDIA GeForce RTX 3090 or more powerful GPUs, such as the RTX 4090 or A100. The results presented in this paper were obtained using an NVIDIA GeForce RTX 3090.

A.2.1 Security, privacy, and ethical concerns

Reproducing Table 3 and Table 4 does not pose any security risks. However, to reproduce the results of the hate speech campaign (Table 6, 8, 9, 10), it is likely to introduce security and ethical concerns, as it involves attacking commercial models such as the Perspective API and OpenAI Moderation API. To mitigate these risks, we recommend that evaluators reproduce the results on the open-source model TweetHate, which is also the default target model provided in the following steps. This ensures that evaluator results do not affect the functionality of commercial models.

A.2.2 How to access

The artifact is available on Zenodo: https://zenodo.org/records/14840447. Given the ethical concerns surrounding our code, which includes attacks against real-world systems, we host these artifacts on Zenodo with the request-access feature enabled.

A.2.3 Hardware dependencies

The artifact requires environments with GPUs. We recommend using an environment with NVIDIA GeForce RTX 3090 or more powerful GPUs, such as the RTX 4090 or A100. The results presented in this paper were obtained using an NVIDIA GeForce RTX 3090.

A.2.4 Software dependencies

All our experiments are tested in a conda environment on Ubuntu 20.04.6 LTS with Python 3.9.0. We provide detailed instructions for evaluators to set up this conda environment (see subsection A.3).

A.2.5 Benchmarks

Two datasets are used in the main experiments: 1) HateBench-Set, the manually-annotated dataset generated by LLMs. 2) MHS dataset[1] as human-written samples.

A.3 Set-up

A.3.1 Installation

Download the artifact from Zenodo, decompress it, and enter the main project directory. Then follow the below commands to build the environment.

```
conda create -n hatebench python=3.9.0
conda activate hatebench
pip install -r requirements.txt
```

Enter the PYTHON environment to download the necessary package.

```
import nltk
nltk.download('averaged_perceptron_tagger_eng')
exit()
```

A.3.2 Basic Test

```
python basic_test.py
```

Output should be

```
Transformers loaded successfully!
Detoxify works! output: {...}
TextAttack module initialized successfully!
OpenAttack loaded successfully!
Pandas & NumPy work!
Basic test completed!
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We show that hate speech detectors perform inconsistently on LLM-generated and human-written samples. This is proven by the experiment (E1, E2) described in Section 4, with the results shown in Tables 3 and 4.

(C2): Adversarial hate campaigns can bypass detectors. This is proven by the experiment (E3) described in Section 5.4, whose results are reported in Table 6.

(C3): Model stealing can be used to enhance stealthy hate campaigns. This is proven by the experiment (E4, E5) described in Section 5.5, whose results are illustrated in Tables 8, 9, and 10.

[1] https://huggingface.co/datasets/ucberkeley-dlab/measuring-hate-speech.

A.4.2 Experiments

(E1): Performance on LLM-generated samples *[1 compute-minute]:*
How to: Run the script to get the results.
Preparation: None
Execution: Run the following command in the terminal from the project's root directory.

```
python measurement/
    calculate_detector_performance.py
```

Results: The results will be printed directly on the terminal. It should be similar to Table 3.

(E2): F1-score on LLM-generated and human-written text. *[2 compute-minutes]:*
How to: Run the script to calculate the performance of hate speech detectors.
Preparation: None
Execution: Run the following command in the terminal from the project's root directory.

```
python measurement/
    calculate_detector_LLM_performance.py
```

Results: The results will be printed directly on the terminal. It should be similar to Table 4.

(E3): Adversarial hate campaign *[30 compute-minutes]:*
How to: Execute the attack scripts to reproduce the results of TweetHate presented in Table 6. Results will be automatically stored in the *./logs/* directory. The naming convention for the log files is *adv_hate_campaign_{target_model}_{attack}.log*. For example, to check the results of *TextFooler* attack on *TweetHate* model, refer to the log file named *adv_hate_campaign_TweetHate_TextFooler.log* and the results are printed at the end of the log.
Preparation: None
Execution: Run the script.

```
cd hate_campaign
bash scripts/run_adversarial_hate_campaign.sh
```

Results: The results are expected to be comparable to those for TweetHate in Table 6. Note that minor deviations may arise due to the inherent stochasticity of machine learning algorithms and differences in GPU performance.

(E4): Steal the target model *[2 compute-hours]:*
How to: Run the script and check the results.
Preparation: None
Execution: Run the script.

```
nohup python model_stealing.py --target_model
    TweetHate --surrogate_model roberta  > ./
    logs/TweetHate_roberta.log &
nohup python model_stealing.py --target_model
    TweetHate --surrogate_model bert  > ./
    logs/TweetHate_bert.log &
```

Results: The results are expected to be comparable to those for TweetHate in Table 8.

(E5): Stealthy hate campaign (black-box + white-box) *[60 compute-minutes]:*

How to: Execute the attack scripts to reproduce the results of TweetHate presented in Tables 9 and 10. Results will be automatically stored in the *./logs/* directory.

Preparation: None

Execution: Run the script.

```
bash scripts/run_stealthy_hate_campaign.sh
```

Results: The results are expected to be comparable to those for TweetHate in Tables 9 and 10. Attack *textfooler* refers to the black-box attack (Table 9). Attack *textfooler_gradient* is the white-box attack (Table 10). The results are printed at the end of the log.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: EmbedX: Embedding-Based Cross-Trigger Backdoor Attack Against Large Language Models

Nan Yan[1], Yuqing Li[1], Xiong Wang[2], Jing Chen[1], Kun He[1], and Bo Li[3]
[1]Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University
[2]School of Computer Science and Technology, Huazhong University of Science and Technology
[3]Department of Computer Science and Engineering, Hong Kong University of Science and Technology

A Artifact Appendix

A.1 Abstract

This artifact accompanies the paper *"EmbedX: Embedding-Based Cross-Trigger Backdoor Attack Against Large Language Models"*.

Existing backdoor attacks against large language models (LLMs) predominantly focus on single-trigger mechanisms, while ignoring the variations in different users' responses to the same trigger, thus often resulting in undermined attack effectiveness. In this work, we propose EmbedX, a novel and efficient cross-trigger backdoor attack that operates in the embedding space of LLMs. EmbedX employs a *soft trigger* mechanism that is activated through token fuse, enabling the injection of backdoors.

This artifact consists of the following components:

- Test Datasets: We evaluate EmbedX across a diverse set of tasks, including binary classification (SST-2, IMDB, Twitter), multi-class classification (Emotion), and instruction tuning for text generation (Alpaca).
- EmbedX Implementation: The artifact provides implementations for the three core stages of the EmbedX attack: (1) soft trigger generation at the embedding layer, (2) latent adversarial backdoor injection under dual stealthiness constraints, and (3) alignment of multiple token embeddings to form an effective cross-trigger that activates the backdoor.
- EmbedX Evaluation: We provide comprehensive testing scripts to reproduce the experimental results reported in the paper. These scripts are configured for four different target LLMs and are designed to evaluate the effectiveness of the cross-trigger attack in a fully reproducible manner.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. All datasets and models used in this work are publicly available and are intended strictly for academic research purposes. We explicitly do not condone, promote, or facilitate the deployment of backdoored LLMs in real-world applications. The intent of this research is to raise awareness of potential vulnerabilities in LLMs and to support the development of more robust defense mechanisms. Additionally, all trigger designs used in our experiments are culturally neutral and do not contain any sensitive, offensive, or harmful content.

A.2.2 How to access

We provide a stable reference via Zenodo (please refer to the latest version) at https://doi.org/10.5281/zenodo.15609883 The artifact can also be accessed in Github by link https://github.com/lunan0320/EmbedX.

A.2.3 Hardware dependencies

This artifact does not require any specialized hardware beyond standard GPU computing resources. For optimal performance, we recommend using either 4 NVIDIA RTX 4090 GPUs (24 GB VRAM per GPU) or a single NVIDIA A100 GPU (80 GB VRAM). Our experiments were conducted on a system running Ubuntu 20.04.3 LTS, equipped with 128 AMD EPYC 7532 CPU cores and 503 GB of RAM.

Compatibility with other GPU configurations is not guaranteed and may require adjustments.

A.2.4 Software dependencies

The artifact requires the following software or packages:

- Python 3.9.12 or above, CUDA 12.0.
- Core Python libraries: `torch`, `transformers`, `datasets`, `peft`, and `accelerate`.

All additional dependencies are listed in the accompanying `requirements.txt` file.

A.2.5 Benchmarks

See below.

A.3 Set-up

A.3.1 Installation

The artifact can be obtained as a zip compressed archive on Zenodo or cloned from the Github repository.

To install the required packages for running EmbedX, execute:

```
pip install -r requirements.txt
```

HuggingFace credentials: Some models used in this project are hosted on HuggingFace and may require authentication to download.

1.Install the HuggingFace Hub tools:

```
pip install huggingface_hub
```

2.Login to your HuggingFace account:

```
huggingface-cli login
```

This command will prompt you to enter your HuggingFace access token.

Next, grant executable permissions to the shell scripts in the `run/` directory:

```
chmod +x run/*.sh
```

To download the datasets, run:

```
./run/0-data.sh
```

To download the required models, run:

```
./run/0-models.sh
```

For additional setup details and usage instructions, please refer to the `README` file in the artifact repository.

A.3.2 Basic Test

This test validates the full EmbedX attack pipeline, focusing on the completeness of its components and the correctness of their functionality. The procedure involves the following steps:

- Step 0: Fine-tune a clean model.
- Step 1: Generate a soft trigger.
- Step 2: Inject a latent adversarial backdoor.
- Step 3: Align multiple token embeddings with the soft trigger.

To simulate the entire attack process using a reduced dataset (approximately 1% of the full training set), run the following commands to execute Embedx step 1, step 2, step 3:

```
./run/0-basic_test.sh
```

Logs will be saved to `/logs/0-basic_test.log`. Upon completion, the outputs of each step can be found in the following directories:

- The generated soft trigger (Step 1): `/trigger_save/`
- The backdoored model after injection (Step 2): `/output/`
- The final backdoored model after token-fuse optimization (Step 3): `/soft_model/`

This run is intended solely to verify that all major components are present and functioning as expected.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): EmbedX achieves superior attack effectiveness and efficiency across four datasets and target LLMs. This claim is supported by experiment (E1), described in Section 4.2, with results presented in Table 2 and Figure 4.

(C2): EmbedX enables fine-grained cross-trigger backdoor attacks using diverse token triggers. This claim is supported by experiments (E2) and (E3), with results shown in Table 3.

A.4.2 Experiments

This section outlines the experimental procedures. You can find the log files from our experiments conducted on a single A100 GPU or 4 RTX 4090 GPUs in the `saved_logs/` directory.

It's worth noting that in all pipeline scripts, we use `pkill` after each step to release GPU memory and prevent *out-of-memory (OOM)* errors in subsequent steps. If you prefer not to do this, you can comment out the other steps in the pipeline script and run only one step at a time.

(E1): [Overall performance comparison] [5 human-minutes + 1.5-3.5 compute-hours]: This experiment evaluates backdoored models generated using rare words and intentional misspellings as token triggers. We assess model utility, attack effectiveness, and attack efficiency using clean test accuracy (CTA), attack success rate (ASR), and training time (Time), respectively.

How to: Please refer to `BLOOM-7b@SST-2`, `Llama2-7b@Twitter`, and `Llama3-8b@Emotion` under the README file for detailed instructions.

Execution: First, run the pipeline script `./run/{number}-pipeline_{model_name}_{dataset_name}.sh` to generate the backdoored model. Then, run the test script to evaluate the performance `./run/{number}-test_{model_name}_{dataset_name}.sh`.

Results: The resulting soft triggers are saved in the `trigger_save/` directory, the soft-triggered backdoored models in `output/`, and the token-fusion-based backdoored models in `soft_model/`. Logs for both training and evaluation are stored in the `logs/` folder.
Summary: Then, please run the command `python parase_log.py logs/{your_log}.log` to parse the log file. You can then find the generated `.csv` and `.md` summary files in the `log/` folder to view the test results table.

(E2): [Cross-style attack] [2 human-minutes + 1.5 compute-hour]: This experiment evaluates the backdoor performance with domain-specific token triggers in instruction-tuned models.
How to: See `Gemma2-9b@Alpaca` under the README.
Execution: First, generate backdoored model with the command `./run/4-pipeline_gemma2_alpaca.sh`. Then, evaluate the performance with command `./run/4-test_gemma2_alpaca.sh`.
Results: As in (E1), the outputs follow the same directory structure. Logs are saved to: `/logs/4-pipeline_gemma2_alpaca.log` and `/logs/4-test_gemma2_alpaca.log`.
Summary: As in (E1), run the command `python parase_log.py logs/{your_log}.log` to parse the log file and view the test results table.

(E3): [Cross custom trigger attack] [2 human-minutes + 2 compute-minutes]: This experiment evaluates the ability of EmbedX to generalize to arbitrary custom token fuses without additional training.
How to: See `Cross New Trigger Attack` under the README file.
Execution: First, remove any existing model files in `soft_model/`. Run the following command to establish the mapping between the new token fuses (e.g., `"loko, th1s, quizzaciously"`) and the soft trigger:
`./run/5-token_fuse_bloom_sst2.sh`
Run the following script to evaluate the attack. The new token fuses will effectively trigger the backdoor behavior without requiring any additional training.
`./run/5-test_token_fuse_bloom_sst2.sh`
Results: The updated backdoored model will be saved in `soft_model/`, and logs for embedding alignment and evaluation are written to `logs/5-token_fuse_bloom_sst2.log` and `logs/5-test_token_fuse_bloom_sst2.log`, respectively.
Summary: As in (E1), run the command `python parase_log.py logs/{your_log}.log` to parse the log file and view the test results table.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Game of Arrows: On the (In-)Security of Weight Obfuscation for On-Device TEE-Shielded LLM Partition Algorithms

Pengli Wang[1], Bingyou Dong[2], Yifeng Cai[1], Zheng Zhang[2], Junlin Liu[1], Huanran Xue[2], Ye Wu[2], Yao Zhang[2], and Ziqi Zhang[3,*]

[1]*MOE Key Lab of HCST (PKU), School of Computer Science, Peking University*
[2]*ByteDance*
[3]*University of Illinois Urbana-Champaign*
[*]*Corresponding Author*

A Artifact Appendix

A.1 Abstract

Utilizing Trusted Execution Environments (TEEs) to protect Large Language Models (LLMs) on users' devices is a practical solution for model owners. To alleviate the computation burden on TEEs, researchers have proposed TEE-Shielded LLM Partition (TSLP) to offload heavy computation layers to co-operating untrusted GPUs, while lightweight layers are shielded in TEE. TSLP utilizes various lightweight obfuscation schemes to protect offloaded weights from various attacks meanwhile not introducing large computation overhead. However, existing lightweight obfuscation algorithms have one vital vulnerability in common: the direction similarity of obfuscated vectors. In this paper, we propose a novel attack, ARROWMATCH, that utilizes direction similarity to recover obfuscated private weights. To achieve this, ARROWMATCH compares direction distances between obfuscated model weights and public pre-trained model weights. To mitigate this vulnerability, we propose a novel obfuscation scheme, ARROWCLOAK, which leverages lightweight matrix-vector multiplication to protect vector directions and private weights. We evaluate ARROWMATCH and ARROWCLOAK on four representative LLMs, using seven datasets, along with five obfuscation schemes. The results show that ARROWMATCH can break the protection of all existing lightweight obfuscation schemes with high accuracy (similar to no protection) and effectively recover the private weights (with over 98% accuracy). In addition, ARROWCLOAK can effectively defend against ARROWMATCH (6.5× better than state of the art) and protect direction information by increasing the direction distance over 900×. We also evaluate the performance of ARROWCLOAK on a real-world Intel SGX device and show that ARROWCLOAK can reduce total overhead by 2.83× compared to shield-the-whole baseline.

A.2 Description & Requirements

We have carefully read the ethics considerations discussions in the conference call for papers, the detailed submission instructions, and the guidelines for ethics documents. This research focuses on the study of deep learning models and Trusted Execution Environments (TEEs) for enhancing secure and efficient computing. The datasets employed in this study were curated from publicly available resources that explicitly permit research use. Similarly, the deep learning models explored in this research are widely used in the academic community. We guarantee that our artifact contains no destructive steps, and executing our code will cause no damage to the host machine. Moreover, this study does not involve experiments on live systems, human participants, or data that could be linked to personally identifiable informa-tion (PII). Furthermore, no terms of service were violated, and the research adheres strictly to both the "Respect for Persons" and "Respect for Law and Public Interest" principles outlined in The Menlo Report.

A.2.1 How to access

Aligning with the open science policy, we release our code on [1, 2]. The code repository contains implementations of ARROWMATCH and ARROWCLOAK, along with corresponding scripts, across several model architectures(ViT, BERT, and GPT). By following the instructions in the code repository's README file, reviewers can reproduce the attack and defense performance in our paper.

A.2.2 Hardware dependencies

Our experiments are executed on 2 × NVIDIA RTX A6000 GPUs (48G). To ensure reproducibility, we recommend reviewers utilize identical or comparable GPUs for model training and evaluation.

A.2.3 Software dependencies

The code execution primarily relies on Python libraries for machine learning and numerical computing, such as torch, transformers, and scipy. To simplify environment configuration for reviewers, we have provided setup scripts (install1.sh and install2.sh) that automate the experimental environment installation.

A.2.4 Benchmarks

We employed four deep learning models: ViT-Base, BERT-Base, GPT-2, and GPT-2-XL, which are open-source models widely used in the academic community. Our experiments were conducted on seven datasets: CIFAR10, CIFAR100, FOOD101, MNLI, QQP, SST-2 and QNLI, where the first three are standard image classification datasets and the latter four are selected from the classic GLUE benchmark.

A.3 Set-up

A.3.1 Installation

We have prepared environment configuration scripts for reviewers: install1.sh (for text tasks) and install2.sh (for visual tasks). By following the instructions in the README within our code repository, reviewers can run these scripts to configure the corresponding experimental environment and activate it for subsequent training or testing tasks.

A.3.2 Basic Test

We have prepared environment test scripts for reviewers in the scripts2 directory: basic_test1.sh (for environment1) and basic_test2.sh (for environment2). By following the instructions in the README within our code repository, reviewers can run these scripts to test the corresponding experimental environment.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): ARROWMATCH can effectively recover the lightweight obfuscation algorithms and achieve a high attack performance. The attack performance is averagely over 1.6× higher than the black-box baseline. This is proven by the experiment described in section 7.2 whose results are reported in Table 3.

(C2): ARROWCLOAK can effectively protect the model against ARROWMATCH. The attack performance is only 1.10× higher than Shield-Whole and is over 6× better than the best prior obfuscation algorithms. This is proven by the experiment described in section 7.4 whose results are reported in Table 5.

A.4.2 Experiments

(E1): *[Fine-tune Phase] [10 human-minutes + 30 compute-hour]:*
Preparation: Check that the scripts exist in the scripts/ and scripts2/ directories, then locate the *train private model* section under Experiments in the README.
Execution: Execute the following scripts sequentially according to the README instructions to fine-tune ViT-Base, BERT-Base, GPT2-Base, and GPT2-XL across multiple datasets, with expected runtimes of 1.5h, 7.5h, 9h and 12h respectively.

```
./scripts2/vit_trains.sh
./scripts2/bert_trains.sh
./scripts2/gpt2_trains.sh
./scripts2/gpt2_xl_trains.sh
```

Results: The fine-tuned results will be stored in the results/train_results and results/tsqp_results directories. The performances are expected to be similar to the results in **White-box column in Table 3**. Please rename the final checkpoint of the BERT and GPT2 results to *final_checkpoint* to conduct subsequent attack and defense experiments.

(E2): *[ARROWMATCH Phase] [10 human-minutes + 8 compute-hour]:*
Preparation: Locate the *Try ARROWMATCH* section under Experiments in the README.
Execution: Execute the following scripts sequentially according to the README instructions to implement ARROWMATCH on ViT-Base, BERT-Base, GPT2-Base, and GPT2-XL across multiple datasets, with expected runtimes of 2.5h, 1.7h, 2h and 1h respectively.

```
./scripts2/vit_arrowmatchs.sh
./scripts2/bert_arrowmatchs.sh
./scripts2/gpt2_arrowmatchs.sh
./scripts2/gpt2_xl_arrowmatchs.sh
```

Results: The ARROWMATCH results will be stored in the results/arrowmatch_results directory. The performances are expected to be similar to the results in **corresponding obfuscation method's column in Table 3**. Please rename the final checkpoint of the BERT and GPT2 results to *final_checkpoint* to conduct subsequent evaluation experiments.

(E3): *[ARROWCLOAK Phase] [10 human-minutes + 2 compute-hour]:*
Preparation: Locate the *Try ARROWCLOAK* section under Experiments in the README.
Execution: Execute the following scripts sequentially according to the README instructions to implement ARROWCLOAK on ViT-Base, BERT-Base, GPT2-Base, and GPT2-XL across multiple datasets, with expected runtimes of 30 minutes,30 minutes, 30 minutes and 30 minutes respectively:

```
./scripts2/vit_arrowcloaks.sh
./scripts2/bert_arrowcloaks.sh
./scripts2/gpt2_arrowcloaks.sh
./scripts2/gpt2_xl_arrowcloaks.sh
```

Results: The ARROWCLOAK results will be stored in the results/arrowcloak_results directory. The performances are expected to be similar to the results in **ARROWCLOAK column in Table 5**. Please rename the final checkpoint of the BERT and GPT2 results to *final_checkpoint* to conduct subsequent evaluation experiments.

(E4): *[Evaluation Phase] [10 human-minutes + 3 compute-hour]:*
Preparation: Locate the *Evaluate the results* section under Experiments in the README.
Execution: Execute the following scripts sequentially according to the README instructions to implement evaluation of ARROWMATCH and ARROWCLOAK on ViT-Base, BERT-Base, GPT2-Base, and GPT2-XL across multiple datasets, with expected runtimes of 45 minutes,45 minutes, 45 minutes and 45 minutes respectively:

```
./scripts2/evaluate_models_vit.sh
./scripts2/evaluate_models_bert.sh
./scripts2/evaluate_models_gpt2.sh
./scripts2/evaluate_models_gpt2_xl.sh
```

Results: The evaluation results will be stored in the evaluation_results directory. Reviewers can locate corresponding results in the evaluate_results/model_name/dataset_name directories, where **whitebox_results and blackbox_results represent the rightmost two columns of baselines in Figure 3; recover_results records the recovery performance of the ARROWMATCH, corresponding to the *our attack* column in Figure 3, while arrowcloak_results documents ARROWCLOAK performance, aligning with the results in column ARROWCLOAK of Figure 5**.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] Github link. https://github.com/qsxltss/Game-of-Arrows, 2025.

[2] Zenodo link. https://zenodo.org/records/15695418, 2025.



USENIX Security '25 Artifact Appendix: Refusal Is Not an Option: Unlearning Safety Alignment of Large Language Models

Minkyoo Song Hanna Kim Jaehan Kim Seungwon Shin Sooel Son
KAIST, South Korea
{minkyoo9, gkssk3654, jaehan, claude, sl.son}@kaist.ac.kr

A Artifact Appendix

A.1 Abstract

This artifact provides the full implementation, experimental pipeline, and datasets used in our paper, which studies adversarial unlearning attacks that compromise the safety alignment of large language models (LLMs). Our codebase supports reproduction of all main results, including the construction and optimization of malicious unlearning datasets, application of four major unlearning algorithms to state-of-the-art open-source LLMs, and comprehensive evaluation against utility and safety benchmarks. Complete setup, training, and evaluation instructions are provided in the repository.

A.2 Description & Requirements

Hardware and Software Requirements:

- **Hardware**: Two NVIDIA A100 GPUs (80GB each), at least 100GB disk space (about 12GB for each unlearned LLaMA model).
- **OS**: Ubuntu 20.04 LTS.
- **CUDA**: Version 11.8.
- **Python**: 3.10.
- **Memory**: At least 256GB RAM is recommended.

See the `environment.yaml` in the repository for complete dependencies.

Benchmarks: All datasets and evaluation scripts needed to reproduce results are included or auto-downloaded. Benchmarks include harmfulness (AdvBench, HEx-Phi, LLM-LAT) and utility (MMLU, BBH, TruthfulQA, TriviaQA, AlpacaEval). For details, see the README or Section 5.1 of the paper.

A.2.1 Security, privacy, and ethical concerns

- No destructive operations or requirement to disable security mechanisms.
- All PII is synthetic or public; no use of real personal data.
- Models capable of harmful output should **not** be deployed beyond research.
- Compromised models are not distributed in the artifact.

A.2.2 How to access

- **Main repository**: `https://github.com/minkyoo9/Unlearning-safety-alignment`
- **Stable DOI**: `https://doi.org/10.5281/zenodo.16740884`

A.2.3 Hardware dependencies

Two NVIDIA A100 80GB GPUs (or equivalent with sufficient memory for 3B–4B parameter LLMs) are required.

Important: If you change the hardware setup, you must review and potentially update the following configuration and script files to ensure compatibility with your new environment:

- `harmfulness/default_config.yaml`
- `harmfulness/inference_config.yaml`
- `LLaMA-Factory/scripts/default_config.yaml`
- Hyperparameters in `LLaMA-Factory/scripts/full`
- Hyperparameters in `muse_bench/baselines/scripts`
- `harmfulness/run_eval_harmful.sh`

A.2.4 Software dependencies

- Ubuntu 20.04; Python 3.10; CUDA 11.8.
- All additional packages (PyTorch, Huggingface Transformers, etc.) listed in `environment.yaml`.
- Install:

```
conda env create -f environment.yaml
conda activate unlearning
```

A.2.5 Benchmarks

- Harmfulness: AdvBench, HEx-Phi, LLM-LAT
- Utility: MMLU, BBH, TruthfulQA, TriviaQA, AlpacaEval
- Filtering: PII (synthetic), fake news (Kaggle), copyright (Harry Potter)
- All datasets are referenced or provided.

A.3 Set-up

A.3.1 Installation

1. Clone repository:
   `git clone https://github.com/minkyoo9/Unlearning-safety-alignment.git`
   `cd Unlearning-safety-alignment`
2. Install dependencies:
   `conda env create -f environment.yaml`
   `conda activate unlearning`
3. Set your Hugging Face token as instructed in the README.

A.3.2 Basic Test

To verify that the environment supports inference with the target model, perform the steps following the instructions in the README (see "Installation and Setup" and "Scenario I: Step 1-1"). After execution, verify that the output file `outs_llama/out_AdvBench` is generated and contains the model's inference results (i.e., rejection responses). Successful completion of this step confirms that the environment is correctly set up for running inference with the target model.

A.4 Evaluation workflow

A.4.1 Major Claims

- (C1): Adversarial unlearning attacks significantly increase the harmfulness of LLMs by more than 10× while preserving general utility.
- (C2): Malicious unlearning requests can bypass filtering systems, compromising LLM safety.
- (C3): The proposed defense detects, on average, >90% of malicious unlearning attempts, thereby mitigating the impact of such attacks.

A.4.2 Experiments

(E1) Unlearning Attacks on Open-Source LLMs:
< 1 human-hour + < 1 GPU-hour per model.
Steps (Scenario I and II in the README):

- Prepare adversarial unlearning datasets.
- Apply DPO/NPO/GA/TV unlearning via provided scripts.
- Evaluate harmfulness/utility.

Outcome: Significantly increased harmfulness scores with utility preserved (Table 2 and 3 of the paper).

(E2) Bypassing Filtering Systems:
< 30 human-minutes + < 30 GPU-minutes.
Steps (Step 1 of Scenario II in the README):

- Construct merged datasets using two LLM agents.
- Apply filtering/classification scripts.

Outcome: >96% adversarial samples bypass filters (Table 4 of the paper).

(E3) Defense Evaluation:
< 30 human-minutes + < 30 GPU-minutes.
Steps (Mitigation in the README):

- Train/apply the defensive classifier.
- Test detection recall on adversarial datasets.

Outcome: Defense detects most of malicious requests, reducing harmfulness (Table 7 of the paper).

See all operational details and examples in the README.

A.5 Notes on Reusability

- The artifact is modular and can be adapted for alternative LLMs and custom datasets.
- Unlearning, data construction, and mitigation modules are independent.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Narrowbeer: A Practical Replay Attack Against the Widevine DRM

Florian Roudot
Univ Rennes, CNRS, IRISA

Mohamed Sabt
Univ Rennes, CNRS, IRISA

A Artifact Appendix

A.1 Abstract

Streaming services like Netflix, Prime Video, and HBO Max rely on DRM solutions to ward off piracy. By enabling the distribution of encrypted content, DRM systems prevent subscribed users from downloading the streamed content, as well as unauthorized users from having access to it.

Google Widevine, one of the most deployed DRMs, provides a fully software-based solution on desktop platforms to ensure portability.

Following our findings of new flaws in the Widevine license acquisition process, we developed Narrowbeer, a practical replay attack that allows legitimate users to generate never-expiring licenses and enables unauthorized users to reuse these licenses to access premium content without subscription. As an artifact of our paper, we provide all the necessary components to reproduce the attack on free content.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Executing the attack does not present any security risks to the user's device. The exploited vulnerability has been responsibly disclosed to Widevine and fixed on the newer versions of the CDM. The licenses provided are extracted from Shaka Player demonstration website[1] and only allow for the playback of free content hosted on Google's servers. The video playback happens in normal conditions and does not represent any risk to Google's servers.

A.2.2 How to access

The artifact is accessible on https://doi.org/10.5281/zenodo.15525617.

A.2.3 Hardware dependencies

Since the licenses were generated on a computer with an x86-64 CPU, they can only be used on systems with the same architecture.

[1] https://shaka-player-demo.appspot.com/

A.2.4 Software dependencies

The attack can be performed on Linux (verified on Debian 12 and Ubuntu 24.04.2) and Windows (verified on Windows 11 Enterprise 24H2), whether real or virtual machines, with software dependencies varying for each system.

On Linux.

- GCC[2] 13.3.0 and Make[3] 4.3 (any recent version should work.)
- Firefox[4] ESR-128 (any recent version should work.)
- Libavcodec[5] from FFmpeg, as some content is distributed using proprietary codecs.

On Windows.

- Visual Studio 2022[6] 17.14.0 with the *Desktop development with C++* kit (any recent version should work.)
- Firefox[4] 131.0.3. The attack does not work with another version as the provided licenses contain a hash of the binary and its signature.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

On Linux.

1. Install GCC, Make, Firefox ESR and libavcodec using a package manager.
2. Compile the attack executable by running `make` in the `Narrowbeer/Linux/` directory.

[2] https://gcc.gnu.org/
[3] https://www.gnu.org/software/make/
[4] https://www.mozilla.org/firefox/
[5] https://packages.debian.org/bookworm/libavcodec59
[6] https://visualstudio.microsoft.com/

On Windows.

1. Download (https://visualstudio.microsoft.com/downloads/) and install Visual Studio. When asked which additional components to install, select *Desktop development with C++* and continue.
2. Download (https://ftp.mozilla.org/pub/firefox/releases/131.0.3/win64/en-US/) and install Firefox 131.0.3.
3. Double click on `Narrowbeer/Windows/narrowbeer.sln` to open it in Visual Studio then build the solution by going to the *Build* menu and selecting *Build Solution*.

On Firefox.

1. On Linux, type `about:config` in the address bar, then set `dom.ipc.forkserver.enable` to `false`.
2. On Windows, disable the auto-update in Firefox's settings.
3. Make sure Widevine is installed by visiting https://bitmovin.com/demos/drm/ and enabling DRMs if asked.
4. Overwrite the newly installed Widevine CDM with the older vulnerable version by copying the file(s) in `WidevineCDM/<system>/` to Widevine's installation folder `gmp-widevine/<version>/`. This folder can be found by typing `about:profiles` in Firefox's address bar and opening the *Root Directory* of the profile in use.

A.3.2 Basic Test

The easiest way to test that everything is functioning correctly is to perform the first and only experiment **(E1)** of this artifact, described in A.4.2

A.4 Evaluation workflow

A.4.1 Major Claims

Our artifact allows for the reproduction of the Narrowbeer attack on Widevine, achieving three main goals:

(C1): Time Manipulation. We hook Firefox time system/API calls and manage to deceive Widevine into thinking a past time is present.

(C2): Randomness Manipulation. We hook Widevine random system/API calls to fix the random values used in the generation of the license request.

(C3): License Replay. Given a previously received and used license, we can manipulate the Widevine CDM to generate the appropriate license request. This allows an attacker to reuse the license and play protected content without contacting the license server, thus bypassing the need to authenticate on premium streaming services.

A.4.2 Experiments

(E1): [Replay Attack] [A few seconds]: Run the Narrowbeer binary to perform the attack on Widevine and play protected content without contacting the license server.
Preparation: On Linux, configure `ptrace` scope by setting the content of the `/proc/sys/kernel/yama/ptrace_scope` file to `1`. Finally, make sure to close all instances of Firefox.
Execution: On Linux, open a terminal and run:
`./narrowbeer firefox-esr`
On Windows, open a terminal and run:
`./narrowbeer <path/to/firefox.exe>`
On both systems, once Firefox is opened, browse to `file://<artifact_root>/Website/index.html`. Click on *Browse* and select one of the license files available in the `Licenses/<system>/` directory, then click on *Play*.
Results: The video should start playing. If the video doesn't stop after a few seconds, it means that Narrowbeer successfully hooked Firefox **(C1)** and Widevine **(C2)**, thus forcing the CDM to generate a fixed request compatible with the reused license **(C3)**. Please note that Firefox needs to be closed and the Narrowbeer binary ran again to perform the attack another time.
If the video stops after a few seconds, the attack failed. Our script can print valuable information in Firefox's console to help troubleshoot the setup. Moreover, the attack on Windows does not work every time and might need to be rerun a few times.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Lancet: A Formalization Framework for Crash and Exploit Pathology

A Artifact Appendix

We aim to apply for Available, Functional and Reproduced Badges.

A.1 Abstract

This artifact is applying for an Artifacts Available Badge, an Artifacts Functional Badge, and a Results Reproduced Badge. For the Available badge, the source code and scripts can be found at the Zenodo [10.5281/zenodo.15611790]. To make it easier for evaluators to reproduce the experiments, we have provided Docker containers via GitHub Packages with public access. Ongoing updates and further development will be released through the same GitHub repository. For the Functional and Reproduced badges, we will use CVE-2019-6977, House of Einherjar, CVE-2024-41965, CPV15, cases in Juliet Test Suite, and CVE-2023-33476, to demonstrate workflows of root cause diagnosing and exploit primitives identification. All experiments are expected to be conducted in Docker container.

A.2 Description & Requirements

In this section, we first describe whether reproducing our artifacts will risk the evaluator's machine security, followed by approaches to accessing our artifacts. Then, we describe hardware dependencies and software dependencies before listing the benchmarks.

A.2.1 Security, privacy, and ethical concerns

`Lancet` analyzes buggy or exploitable softwares' behavior using Intel Pintools. It instruments programs at runtime without interfering their original behavior. To address any concerns about executing code, we provide Docker images that do not require privileged flags. Evaluators can pull these images from GitHub Container Registry and reproduce our results in an isolated environment, ensuring the host machine's safety and privacy (see Section A.2.1). Moreover, all exploits and bug instances have been modified to be harmless and cannot escape from the Docker container. As a result, there are no security, privacy, or ethical risks for the community.

A.2.2 How to access

The artifact can be accessed at `https://zenodo.org/records/15611790`.

A.2.3 Hardware dependencies

To reproduce `Lancet` and obtain reasonably precise results, we recommend the following minimum hardware configuration: (1) an Intel CPU of similar performance to the i5-13600KF, (2) at least 16GB of RAM, and (3) a minimum of 50GB of disk space with a Docker-enabled environment.

A.2.4 Software dependencies

We recommend conducting the evaluation on the Ubuntu Linux distribution, preferably version 22.04 Desktop with Docker installed, which matches the development environment used for `Lancet`. The specific libc version associated with the distribution affects the precise memory layout of the bugs used in our artifact evaluation. While we provide a Docker image to eliminate environmental discrepancies and improve reproducibility, we conservatively advise performing all reproduction efforts within the suggested environment to ensure maximum consistency and accuracy.

A.2.5 Benchmarks

The exploits and bug instances used as test cases are embedded within the provided Docker images and are also publicly accessible followig the references provided in the paper. To evaluate runtime overhead, we use two modes: a baseline mode (without our reasoning logic) and a full mode. This comparison allows us to isolate the overhead introduced by our approach from that of the underlying Pin-based instrumentation. The difference in execution time between these two modes represents the overhead reported in the paper.

A.3 Set-up

In this section, we focus on how to test `Lancet` using the Docker images we provided.

A.3.1 Installation

Install Docker using your system's package manager (*e.g.*, apt or yum), then pull all images from the GitHub repository: https://github.com/a85tract/Lancet.

A.3.2 Basic Test

N/A

A.4 Evaluation workflow

Evaluate whether `Lancet` identifies the root cause with the the correct stdout output. More specific details, including root cause references for each bug and exploit, can be found in the GitHub repository.

A.4.1 Major Claims

(C1): `Lancet` can accurately analyze the given bugs and exploits.

(C2): The time consumption of each analysis task is in line with what is claimed in Table 1, Table 2 and Table 5 of the paper.

A.4.2 Experiments

(E1): [Bug CPV15] [10 human-minutes + 10 compute-minutes + 5GB disk]:
Preparation: `docker pull ghcr.io/a85tract/lancet:cpv15`
Execution: (1) `docker run ghcr.io/a85tract/lancet:cpv15 baseline` (2) `docker run ghcr.io/a85tract/lancet:cpv15 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should report: ① allocation's stacktrace of corrupted memory buffer ② cell owner id and value owner id of target memory cell ③ pointer untrusted dereference

(E2): [Bug CVE-2024-41965] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:cve_2024_41965`
Execution: (1) `docker run ghcr.io/a85tract/lancet:cve_2024_41965 baseline` (2) `docker run ghcr.io/a85tract/lancet:cve_2024_41965 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect double free and log the exact instruction numbers executed prior to the double free bug.

(E3): [Bug CVE-2019-6977] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:cve_2019_6977`
Execution: (1) `docker run ghcr.io/a85tract/lancet:cve_2019_6977 baseline` (2) `docker run ghcr.io/a85tract/lancet:cve_2019_6977 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should: ① detect out-of-bounds writes by logging ADD instructions that violate memory safety rules within paper's reasoning model ② allocation's stacktrace of corrupted memory buffer

(E4): [Exploit CVE-2023-33476] [15 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:cve_2023_33476`
Execution: (1) In order to reproduce this experiment more smoothly, we recommend disable ASLR on the host (outside docker container) by: `echo 0 > /proc/sys/kernel/randomize_va_space` (need root permission). This operation is reversible by echo 2 to the same file. (2) Launch terminal A, run: `docker run -it ghcr.io/a85tract/lancet:cve_2023_33476`, then run: `cd /app/src/minidlna-git` inside the container (3) Launch another terminal B, use docker exec to enter the same contianer, then run: `cd /app/exploits` (4) In terminal A, run: `/app/pin/pin -t /app/lancet.so -- ./minidlnad -R -f minidlna.conf -d`, and wait until no new logs are printed (about 1 minute) (5) run: `python3 tpoison-nopie-x64_reverse-shell.py 127.0.0.1 --system_addr 0x7ffff63a52f0 --got_addr 0x45e150`, in terminal B
Optional: We have also prepared an ASan version; detailed steps can be found in the GitHub README.
Results: `Lancet` should report: ① memmove violation ② logs such as: "[exploit primitive] allocation in .got.plt region", indicating memory overlapping ③ logs such as: "[exploit primitive] write in .got.plt region" indicates expolit is trying to overwrite GOT table

(E5): [Juliet Test Suite & How2heap] [30 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:juliet_how2heap`
Execution: (1) `docker run ghcr.io/a85tract/lancet:cve_2019_6977 <tool> <target> <case>` tool options: `Asan`, `lancet`, `baseline`, target options: `how2heap`, `juliet`, how2heap case options: `fastbin_reverse_into_tcache`, `house_of_einherjar`, `poison_null_byte`, juliet case options: `CWE121_Stack_Based_Buffer_Overflow__CWE129_fgets_01-bad`, `CWE416_Use_After_Free__new_delete_class_01-bad`, ... (2) Hacknote: run `docker run -it --entrypoint /bin/bash ghcr.io/a85tract/lancet:juliet_how2heap`, then run commands inside the docker `cd hacknote; python3`

`exp.py`. Wait until no more output is generated and receive `Your choice :$`, then input 4 with enter.
Results: (1) Each run reports total time consumption; the difference between baseline and lancet indicates the time taken by `Lancet`. (2) Due to space constraints, we'll place the remaining content in the GitHub repo's README file.

(E6): [Bug OSV-2024-204] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:osv_2024_204`
Execution: (1) `docker run ghcr.io/a85tract/lancet:osv_2024_204 baseline` (2) `docker run ghcr.io/a85tract/lancet:osv_2024_204 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect DANGLINGPTR and log the exact instruction numbers executed prior to the use of freed pointer.

(E7): [Bug FFmpeg, #11228] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:ffmpeg_11228`
Execution: (1) `docker run ghcr.io/a85tract/lancet:ffmpeg_11228 baseline` (2) `docker run ghcr.io/a85tract/lancet:ffmpeg_11228 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect NLLPTRDEREF/UNTRUSTEDPTRDEREF.

(E8): [Bug FFmpeg, #10749] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:ffmpeg_10749`
Execution: (1) `docker run ghcr.io/a85tract/lancet:ffmpeg_10749 baseline` (2) `docker run ghcr.io/a85tract/lancet:ffmpeg_10749 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect NLLPTRDEREF .

(E9): [Bug OSV-2023-1276] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:osv_2023_1276`
Execution: (1) `docker run ghcr.io/a85tract/lancet:osv_2023_1276 baseline` (2) `docker run ghcr.io/a85tract/lancet:osv_2023_1276 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect DANGLINGPTR and log the exact instruction numbers executed prior to the use of freed pointer.

(E10): [Bug CVE-2024-43374] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:cve_2024_43374`
Execution: (1) `docker run ghcr.io/a85tract/lancet:cve_2024_43374 baseline` (2) `docker run ghcr.io/a85tract/lancet:cve_2024_43374 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect DANGLINGPTR and log the exact instruction numbers executed prior to the use of freed pointer.

(E11): [Bug GPAC, #2701] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:gpac_2701`
Execution: (1) `docker run ghcr.io/a85tract/lancet:gpac_2701 baseline` (2) `docker run ghcr.io/a85tract/lancet:gpac_2701 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect DANGLINGPTR and log the exact instruction numbers executed prior to the use of freed pointer.

(E12): [Bug GPAC, #2583] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:gpac_2583`
Execution: (1) `docker run ghcr.io/a85tract/lancet:gpac_2583 baseline` (2) `docker run ghcr.io/a85tract/lancet:gpac_2583 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect DANGLINGPTR and log the exact instruction numbers executed prior to the use of freed pointer.

(E13): [Bug PHP, #16595] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:php_16595`
Execution: (1) `docker run ghcr.io/a85tract/lancet:php_16595 baseline` (2) `docker run ghcr.io/a85tract/lancet:php_16595 lancet`
Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect DANGLINGPTR and log the exact instruction numbers executed prior to the use of freed pointer.

(E14): [Bug OSV-2024-96] [10 human-minutes + 10 compute-minutes + 5GB disk]
Preparation: `docker pull ghcr.io/a85tract/lancet:osv_2024_96`
Execution: (1) `docker run ghcr.io/a85tract/lancet:osv_2024_96 baseline` (2)

`docker run ghcr.io/a85tract/lancet:osv_2024_96 lancet`

Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect CROSSBOUNDARY.

(E15): [Bug PHP, #76041] [10 human-minutes + 10 compute-minutes + 5GB disk]

Preparation: `docker pull ghcr.io/a85tract/lancet:php_76041`

Execution: (1) `docker run ghcr.io/a85tract/lancet:php_76041 baseline` (2) `docker run ghcr.io/a85tract/lancet:php_76041 lancet`

Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect NLLPTRDEREF.

(E16): [Bug CVE-2004-1287] [10 human-minutes + 10 compute-minutes + 5GB disk]

Preparation: `docker pull ghcr.io/a85tract/lancet:cve_2004_1287`

Execution: (1) `docker run ghcr.io/a85tract/lancet:cve_2004_1287 baseline` (2) `docker run ghcr.io/a85tract/lancet:cve_2004_1287 lancet`

Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect CROSSBOUNDARY.

(E17): [Bug CVE-2007-1001] [10 human-minutes + 10 compute-minutes + 5GB disk]

Preparation: `docker pull ghcr.io/a85tract/lancet:cve_2007_1001`

Execution: (1) `docker run ghcr.io/a85tract/lancet:cve_2007_1001 baseline` (2) `docker run ghcr.io/a85tract/lancet:cve_2007_1001 lancet`

Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect CROSSBOUNDARY.

(E18): [Bug CVE-2012-2386] [10 human-minutes + 10 compute-minutes + 5GB disk]

Preparation: `docker pull ghcr.io/a85tract/lancet:cve_2012_2386`

Execution: (1) `docker run ghcr.io/a85tract/lancet:cve_2012_2386 baseline` (2) `docker run ghcr.io/a85tract/lancet:cve_2012_2386 lancet`

Results: (1) Each run reports total time consumption; the difference between the two indicates the time taken by `Lancet`. (2) `Lancet` should detect CROSSBOUNDARY.

A.5 Correction

During the artifact evaluation, we identified two typographical errors in Table 1 of the paper. The FCS report for the PHP #76041 should be `NLLPTRDEREF`, and the report for the FFmpeg, #10749 should also be `NLLPTRDEREF`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Synthesis of Code-Reuse Attacks from `p-code` Programs

Mark DenHoed
University of Oxford

Tom Melham
University of Oxford

A Artifact Appendix

A.1 Abstract

Our artifact is composed of four components:

- `jingle`: a Rust library for modeling the semantics of a fragment of `p-code`, the Intermediate Language of the Ghidra decompiler, in the language of Satisfiability Modulo Theories (SMT).
- `crackers`: a Rust library for synthesizing Code Reuse Attacks (aka. ROP Chains) from a simple `p-code` program using models produced by `jingle` and a conflict-driven search powered by a Boolean Satisfiability (SAT) solver.
- Our evaluation setup comparing the performance of `crackers` to several open-source ROP synthesis tools. We include our raw data, utilities to generate our charts and tables, and the code needed to re-run the entire evaluation.
- A case-study demonstrating the use of `crackers` in synthesizing a ROP chain for CVE-2017-14493: a simple Remote Code Execution vulnerability in `dnsmasq`.

We include standalone copies of `jingle` and `crackers` for the record. While evaluators may install and use these tools, bundled copies of these libraries are used internally by our ROP tool evaluation and case-study. The standalone copies are not necessary to evaluate the claims of the paper. We will indicate instructions for the optional usage of these tools with the heading **`jingle` and `crackers` (optional)**.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifacts pose no risk to evaluator's machines.

A.2.2 How to access

Our artifacts can be accessed via Zenodo using the DOI 10.5281/zenodo.14738160

A.2.3 Hardware dependencies

While our real-world exploitation case-study is dockerized, it requires an ARM CPU due to its usage of GDB and `ptrace`. We tested it on an Apple M1 processor.

We don't require any specific CPU for `jingle`, `crackers`, or our ROP tool evaluation. As many of the evaluated tools use SMT Solvers, we recommend that our ROP tool evaluation be run on a system with at least 32 Gigabytes of RAM. We also recommend that at least 15 gigabytes of disk space reserved for Docker due to the large number of built containers.

A.2.4 Software dependencies

The ROP tool evaluation and the `dnsmasq` case study both run fully inside Docker and make use of Docker Compose, so Docker must be installed. We additionally recommend the Just command runner to simplify the usage of our Docker Compose setup, but this is not mandatory.

`jingle` and `crackers` (optional). Both tools require that the system have both the Ghidra and Z3 to be installed to run. If using a macOS system, the version of Z3 in Homebrew will work; if using a Debian-based linux, we recommend using the build distributed by the Z3 project, as Debian distributes a very outdated version of Z3. Additionally, both these libraries require the Rust compiler to be installed as well as your platform's standard C/C++ compiler/linker. We have used both tools on both x86 and ARM, as well as linux and macOS. The tools should work on Windows systems as well, but we have not verified this outside of a CI pipeline.

A.2.5 Benchmarks

Our ROP tool evaluation uses the ALLSTAR data-set as a source of test binaries. We do not include this dataset in the artifacts due to its size. Instead, our evaluation code retrieves binaries from JHU/APL's hosted version of the data-set as needed. These files are saved temporarily within a docker container and deleted afterwards to minimize the needed disk space. An active internet connection is therefore needed throughout the evaluation.

A.3 Set-up

A.3.1 Installation

ROP Tool Evaluation. In the `crackers_evaluation` folder, verify docker is running with `docker --version` and attempt to build all the images used in the evaluation with `docker compose build`. This will build a container for every ROP tool and algorithm ablation evaluated in the paper. The docker build should succeed with no errors.

`dnsmasq` Case Study (ARM ONLY). In the `dnsmasq_poc/arm64` folder, run `docker compose --profile poc build`. This should build both a container with a vulnerable version of `dnsmasq` and a container named `poc` containing a tool for exploiting this vulnerablity using `crackers`. The docker build should succeed with no errors.

`jingle` and `crackers` (optional). Install Rust, Z3, and Ghidra from the links provided above. Find the root directory for each tool (the highest one with a `Cargo.toml` file inside it). Now, for `jingle`, run `cargo install --path ./jingle --features="bin_features"`. For `crackers`, run `cargo install --path . --features="bin"`. There should now be two new binaries in your path, named `jingle` and `crackers`.

A.3.2 Basic Test

The basic ROP tool evaluation setup can be tested first by executing `just restore` or by manually running the following commands:

```
docker compose down redis
docker run -d --rm --name temp \
       -v crackers_evaluation_cache:/data \
       ubuntu sleep infinity
docker cp evaluation_data.rdb \
       temp:/data/dump.rdb
docker stop temp
docker compose up redis -d
```

This loads our original evaluation data into the testing infrastructure. Our original graphs can be produced by running:

```
docker compose run --rm grapher \
       all-tool.svg combined all-tool
docker compose run --rm grapher \
       ablation.svg combined ablation
```

This should produce two SVG files containing the charts from the paper with our results from (E1) and (E2). Clear the database of our evaluation data by running `just cleanup_db` or by manually running `docker compose exec redis redis-cli flushall`.

To verify that the `dnsmasq` case-study is ready, run `docker compose up dnsmasq` to verify that `dnsmasq` was built and runs.

`jingle` and `crackers` (optional). `jingle` can be easily tested with the following command: `jingle disassemble <path_to_ghidra> x86:LE:64:default 55`. This should produce the output `PUSH RBP`. Replace `disassemble` with `lift` to see the `p-code` associated with this x86 instruction, or with `model` to see `jingle`'s SMT modeling of this `p-code`. To test `crackers`, you can use a sample that is provided in the `crackers` directory. First, edit `crackers/sample_config.toml` and ensure that the `ghidra_path` parameter points to your Ghidra installation. You can then run `crackers synth sample_config.toml` and verify that `crackers` finds a ROP chain.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): `crackers` matches or exceeds the performance of the evaluated state-of-the-art ROP chain synthesis tools across a wide corpus of binaries. This is proven by the experiment (E1) described in Section 5.1 whose results are illustrated in Figure 3 and Table 1.

(C2): `crackers` matches or exceeds the performance of the evaluated algorithm ablations across a subset of our corpus of binaries. This is proven by the experiment (E2) described in Section 5.2 whose results are illustrated in Figure 4 and Table 2.

(C3): `crackers` is capable of synthesizing a working ROP chain within the constraints of a real-world vulnerability. This is proven by the experiment (E3) described in Section 5.3.

A.4.2 Experiments

(E1): ROP Tool Comparison [1 human hour + 2 compute-weeks + 30GB Disk + 32GB RAM]: A comparison between `crackers` and several ROP chain synthesis tools over a large corpus of binaries demonstrating `crackers`' equivalent or superior run-time performance and success-rate.

How to: This evaluation is performed through our docker-based testing infrastructure. Each ROP tool is evaluated in its own docker container with results automatically reported and organized in a shared `redis` database. We recommend executing the tools serially to avoid multiple tools' memory and CPU usage confounding timing measurements. We also provide dockerized tools to export the contents of the database into charts and tables of the kind used in our paper, allowing for easy comparison with our results.

Preparation: No additional configuration is required beyond the items already covered in previous sections.
Execution: This evaluation compares `crackers` to three other ROP tools: `angrop`, `exrop`, and SGC. To run this evaluation, you will run one docker command per tool in series (the commands themselves are explained in more detail in a README in our artifacts). In our evaluation, runtimes of each command ranged from around 12 hours to 10 days.
Results: Once the evaluations have finished, you can run our dockerized charting and statistics utilities to produce a version of Figure 3 and Table 1 derived from your new data. These reporting commands can also be run from another terminal at any time during the experiment. This allows for monitoring of progress and truncated runs. The commands to use these dockerized utilities are explained in more detail in the README.
Due to known malformed data in the ALLSTAR dataset, the evaluation runner will periodically fail to decode a package, indicated by the message "`error decoding response body`". This is expected and does not affect the results of the evaluation.
The test runner may occasionally fail to fetch an ALLSTAR package due to networking issues. We include a command to identify binaries in the `crackers` dataset that are missing from that of another ROP tool. Run `docker compose run --rm stats missing <tool>` to receive a summary of missing binaries for the given tool, as well as docker commands to re-attempt evaluation of those specific test cases.

(E2): Ablation Study [1 human hour + 2 compute-weeks + 15GB Disk + 32GB RAM]: This experiment evaluates `crackers` relative to two algorithmic ablations: `crackers_a` and `crackers_b`.
How to: This experiment uses the same Docker Compose setup as (E1), run over a separate set of Docker containers, each containing an ablation of `crackers`.
Preparation: As in (E1), no additional configuration is required beyond the items already covered in previous sections.
Execution: Like (E1), this experiment is conducted by executing several long-running docker commands. In our evaluation, runtimes of each command ranged from around 12 hours to 10 days.
Results: These results are validated using the same method as in (E1). The chart and table produced by this experiment can be compared to Figure 4 and Table 2 respectively.

(E3): Exploitation Case Study [15 human-minutes + 15GB Disk + 15 compute-minutes (ARM ONLY)]: This experiment demonstrates the use of `crackers` in an exploitation tool targeting `dnsmasq` (CVE-2017-14493). We configure `crackers` to synthesize a ROP chain that cleanly exits the program with a controlled exit code, without throwing any signals.
How to: This evaluation is also performed using Docker Compose-based infrastructure, located in `dnsmasq_poc/arm64`. A vulnerable `dnsmasq` service is started in a container, listening for DHCP6 requests on a non-standard port. Our exploitation tool is run in a separate docker container, targeting the private IPV6 docker network address of the `dnsmasq` container. The container with the exploitation tool uses `crackers` to synthesize a ROP chain, which it uses to construct a DHCP6 packet that it sends to `dnsmasq`. If all works correctly, `dnsmasq` exits with our chosen error code.
Preparation: As in (E1) and (E2), no additional configuration is required beyond the items already covered in previous sections.
Execution: This is most easily executed with the Just command runner. First, start the `dnsmasq` container and connect its `dnsmasq` service to GDB by running `just debug`. Once gdb is attached, run `just poc` in another shell. This will build the exploit tool (if you have not already built the container), and begin synthesis of a ROP chain for the `dnsmasq` binary.
Results: When synthesis completes successfully, it will fire off the DHCP6 packet. You should observe `dnsmasq` exit cleanly without tripping GDB.

A.5 Notes on Reusability

Both `jingle` and `crackers` were written to enable further usage. Both tools are standalone Rust crates, exposing programmatic APIs as well as simple command-line tools.

Our Rust API for ALLSTAR is also written as a standalone Rust crate, and so can easily be used or extended in other applications.

Our ROP tool evaluation setup makes very few assumptions about the tools it runs and could easily be extended with additional ROP tools. It could also be easily adapted to measure run-time performance of any program analysis tool over portions of the ALLSTAR dataset.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis

Yuxi Ling* Gokul Rajiv* Kiran Gopinathan† Ilya Sergey*

*National University of Singapore
†University of Illinois Urbana-Champaign

A Artifact Appendix

A.1 Abstract

This appendix presents the artifact of the paper: "Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis". We first present a detailed description of hardware and software needed to build our framework Doppler and the baseline framework BOPC. We also provide an experiment list and a comprehensive guide to reproduce the experiments with estimated time. Additionally, we discuss how Doppler is reusable for new programs.

A.2 Description & Requirements

This section introduces the necessary requirements to set up the environment for running the artifact and recreating the same experiments as in the paper, including ethical concerns, how to access the artifact, hardware and software dependencies, and benchmarks. We recommend evaluators run the artifact in a Docker container.

A.2.1 Security, privacy, and ethical concerns

There is no security risk for evaluators while executing this artifact in their Docker container. This artifact is fully executed locally without any possible data collection or leakage. All the data used in the experiments are publicly available and do not contain any sensitive information.

A.2.2 How to access

The artifact is available on Github[1]. Two Dockerfiles are provided to build the Docker image for our framework Doppler and the baseline framework BOPC in the Github repository.

A.2.3 Hardware dependencies

A commodity laptop in either ARM or AMD(x86-64) architecture is sufficient to run the artifact. A minimum of 4GB of RAM and 5GB of disk space is recommended. All the experiments in the paper are conducted on a commodity laptop running Ubuntu 22.04 with Intel Core i7 processor, 16GB of RAM, and 512GB of disk space.

A.2.4 Software dependencies

The artifact can be either built from the source code or the Dockerfile. If evaluators choose to build the artifact from the source code, the main dependencies are listed below:

- Ubuntu 22.04 `(Recommended)`
- LLVM 13
- Clang 13
- Z3 4.13.4
- KLEE 3.0

If the evaluators choose to build the artifact from the Dockerfile, the main software required is Docker. A detailed list of environmental dependencies and installation commands is provided in the Dockerfile.

A.2.5 Benchmarks

There are 17 benchmarks used in the paper, including 6 characteristic programs to illuminate Doppler features and 11 real-world programs to show the scalability of it.

A.3 Set-up

This section introduces how to set up the environment and get the executables to run our framework Doppler and the baseline framework BOPC.

A.3.1 Installation

Once the repository is cloned and the Docker is installed, evaluators can build the environment by running the following commands:

- `cd DOPPLER`

[1] https://zenodo.org/records/14718582

- `docker build -t doppler-image .`
- `docker run -it -name my-doppler doppler-image /bin/bash`

In the docker container `my-doppler`, you will get an executable file `DOPPLER/build/doppler`. The same procedure can be applied to the BOPC.

A.3.2 Basic Test

To test if Doppler is correctly installed and executable, direct to the path `DOPPLER/build` and test the command: `./doppler -h`. If the help message is printed, Doppler is correctly installed. You can set up other options and input files to test Doppler further.

To rest if BOPC is correctly installed and executable, direct to the path `BOPC_evaluation` and test the command: `python ./BOPC/source/BOPC.py`. If the warning message complains the input file is missing, BOPC is correctly installed.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Doppler and BOPC work in all benchmark programs. The results are consistent with those shown in Table 4 of our paper.

(C2): The statistics for the grammar synthesised by Doppler in all benchmark programs are consistent with those shown in Table 5 of our paper.

A.4.2 Experiments

(E1): *[Running BOPC demo] [5 human-minutes + 0.5 compute-hour]: run demo programs in BOPC and collect results.*
How to: *Execute the shell script provided in the file `BOPC_evaluation/tb4_bopc.sh`*
Execution: *Detailed instructions are provided in the last section of the file `BOPC_evaluation/README.md`*
Results: *Execution results are stored in a CVS file with the execution time of each program in each attack goal. Succeed tasks will produce a payload file with `.gdb` suffix.*

(E2): *[Running BOPC real] [5 human-minutes + 9 compute-hour]: run real-world programs in BOPC and collect results.*
How to: *Execute the shell script provided in the file `BOPC_evaluation/tb4_bopc.sh` with a different argument.*
Execution: *Detailed instructions are provided in the last section of the file `BOPC_evaluation/README.md`*
Results: *Execution results are stored in a CVS file with the execution time of each program in each attack goal. Three tasks would cause a timeout of 2 hours.*

(E3): *[Running Doppler demo] [5 human-minutes + 1 compute-hour]: run demo programs in Doppler and collect results.*
How to: *Execute the shell script provided in the file `DOPPLER_evaluation/demo_run.sh`.*
Execution: *It would automatically compile all demo programs and execute Doppler. Detailed instructions are provided in the last section of the file `DOPPLER_evaluation/README.md`*
Results: *Output files include a generated DFA , a generated grammar, a compiler (an executable binary), and a trace file. To reproduce the attack payloads described in Table 4 of the paper, evaluators should manually check the grammar and determine wether curtain attacks are possible. To reproduce the grammar statistics, evaluators should check the execution log files and get the statistics.*

(E4): *[Running Doppler real] [5 human-hour + 12 compute-hour]: run real-world programs in Doppler and collect results.*
How to: *Execute the shell script provided in the file `DOPPLER_evaluation/real_run.sh` with different arguments.*
Execution: *It would automatically compile one real program and execute Doppler each time. Detailed instructions are provided in the last section of the file `DOPPLER_evaluation/README.md`*
Results: *Output files include a generated DFA , a generated grammar, a compiler (an executable binary), and a trace file. The procedure to reproduce results described in the paper is the same as (E3).*

A.5 Notes on Reusability

To run a new program with Doppler, users need to do the following preparations:

- Know the position of the vulnerability.
- Add KLEE annotations into the source code and compile it to LLVM IR file.
- Claim corruptable variables in a JSON file.

More detailed instructions for each step can be found in the README file in the Github repository.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: My ZIP isn't your ZIP: Identifying and Exploiting Semantic Gaps Between ZIP Parsers

Yufan You[1], Jianjun Chen[1,2,✉], Qi Wang[1], and Haixin Duan[1,2]

[1]Tsinghua University
[2]Zhongguancun Laboratory

A Artifact Appendix

A.1 Abstract

Our artifact consists of three major components, along with several utility scripts to reproduce the results in the paper:

1. The ZipDiff differential fuzzer designed to discover inconsistencies between multiple ZIP parsers. It has two options to reproduce the ablation study in the paper.

2. Docker images for the 50 tested ZIP parsers. We provide both prebuilt image files and source files to build them.

3. Construction code for ambiguous ZIP files. Based on our findings, we classify the ZIP parsing ambiguities into 14 types and constructed sample ZIP files for each variant of them. We use the construction code to demonstrate how to implement the ambiguities described in the paper. We also use them to measure the types of inconsistencies between each of the 1225 pairs of parsers.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All ZIP files in this artifact are only used to test inconsistencies between ZIP parsers. They do not contain harmful payloads such as malware, and the parsers are isolated in Docker containers, so it is safe to run the experiments.

A.2.2 How to access

Our artifact is available on Zenodo[1] and GitHub[2]. On Zenodo, the source code is provided in `ZipDiff.tar.gz`, and the prebuilt Docker images are provided in the `*.tar.bz2` files.

✉Corresponding author: jianjun@tsinghua.edu.cn
[1]https://doi.org/10.5281/zenodo.15526863
[2]https://github.com/ouuan/ZipDiff/tree/zenodo-v7

A.2.3 Hardware dependencies

We recommend running the experiment on a machine with at least 128GB of RAM and 300GB of disk space. More CPU cores will speed up the experiment. The authors used Intel Xeon Gold 6330 with 112 logical cores and 944GB of RAM.

The memory requirement comes from two sources: First, it runs 50 parsers in parallel that may consume around 60GB of RAM at peak. Second, the unzip outputs can sometimes be very large, and the file I/O operations will be slow if it exceeds the cache size. The memory usage also depends on the number of CPU cores, as the parsers will run in parallel based on the number of available CPUs.

The disk space requirement is also mainly caused by large unzip outputs. The outputs are compressed after each iteration, so the final results will be smaller than the peak usage during a single iteration. The disk space requirement will be much lower if the file system supports transparent compression.

The fuzzer provides a batch size option that can be lowered to reduce the resource requirements. However, the Docker overhead will be significant if the batch size is too small.

A.2.4 Software dependencies

The experiments require a Linux system with Rust, Docker and Docker Compose. Python 3 with `numpy` and `matplotlib` is required for generating the tables and figures. The `zip` command will be helpful in the basic test.

The authors used Ubuntu 23.10 with Linux 6.5.0-44, Rust 1.86.0, Docker 27.1.1 with Docker Compose 2.33.1, and Python 3.13.3 with numpy 2.3.0 and matplotlib 3.10.3.

A.2.5 Benchmarks

The tested ZIP parsers are provided as Docker images. The prebuilt image files are available on Zenodo.

A.3 Set-up

A.3.1 Installation

Follow these steps to set up the dependencies and the artifact:

1. **Install Rust.** Follow the instructions at https://www.rust-lang.org/tools/install to install Rust using rustup. Either install the latest stable version (by default) or 1.86.0 used by the authors.

2. **Install Docker and Docker Compose.** Follow the instructions at https://docs.docker.com/engine/install and https://docs.docker.com/compose/install/linux.

3. **Fetch source code.** Either download `ZipDiff.tar.gz` from Zenodo and extract the archive by `tar xf ZipDiff.tar.gz`, or clone it from GitHub by `git clone https://github.com/ouuan/ZipDiff`.

4. **Compile Rust code.** Run `cargo build --release` in the `zip-diff` directory of the source code.

5. **Build or load Docker images.** Build Docker images of the ZIP parsers by running `tools/prepare.sh`, `cd parsers`, and `sudo docker compose build`. Alternatively, load the prebuilt image files: `for i in *.tar.bz2; do docker load -i "$i"; done`. Most parser versions are fixed in the source files, but a few may change, so it is recommended to load prebuilt images to ensure using the same versions as the paper.

A.3.2 Basic Test

Test parsers. Create a ZIP: `zip -0 a.zip README.md`. Run parsers on it: `tools/run-parsers.sh a.zip`. The parsers will print some logs and exit. The outputs of the parsers will be saved in subdirectories of `evaluation/results/a.zip`. All parsers should extract `README.md` successfully.

Minimal fuzzing. Run `sudo target/release/fuzz -b 10 -s 120` in the `zip-diff` directory. This will run fuzzing for two minutes with only ten samples per batch. The fuzzer will print logs for each iteration. The log text should contain `ok: 50`, indicating that all parsers are working fine. The results will be available at `evaluation/stats.json`, `evaluation/samples` and `evaluation/results`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): ZIPDIFF can discover inconsistencies between the majority of the 50 tested ZIP parsers. The chosen system design outperforms the two other setups in the ablation study. This is proven by the experiment (E1) described in Section 5.1 and 5.3, with results plotted in Fig. 4.

(C2): We identified 14 types of ZIP parsing ambiguities as described in Section 5.2. We constructed sample files corresponding to these ambiguities. All 14 types can cause inconsistencies between some pairs of parsers, and 1221 out of a total of 1225 pairs of parsers are affected by at least one type of ambiguity. This is proven by the experiment (E2) and the number of inconsistency types between each pair of parsers is recorded in Table 4.

A.4.2 Experiments

(E1): Fuzzing and ablation study [5 human-minutes + 15 computer-days + 100GB final / 500GB runtime disk]
Preparation: We default to run five 24-hour fuzzing sessions for each of the three setups with a batch size of 500. For quick verification, the duration of each session (`STOP_SECONDS` in `tools/ablation-study.sh`) and the number of sessions (`TIMES`) can be reduced. To adapt for low memory and disk space, the batch size (`BATCH_SIZE`) can be reduced.
Execution: Run `sudo tools/ablation-study.sh`. It requires root privilege because a normal user might not have the permission to read the unzip outputs.
Results: The stats are saved in `evaluation/stats`. The samples and outputs are saved in the subdirectories of `evaluation/sessions`. Run `python3 tools/fuzz-stats.py evaluation/stats/*` to plot the graph (Fig. 4) at `inconsistent-pair-cdf.pdf`.

(E2): Ambiguous ZIP construction and inconsistency type counting [1 human-minute + 40 computer-minutes]
Execution: Inside the `zip-diff` directory, first run `target/release/construction`, and then run `sudo target/release/count`.
Results: The inconsistency type details are saved at `constructions/inconsistency-types.json`. Run `python3 tools/inconsistency-table.py` to generate the LaTeX parser inconsistency table (Table 4).

A.5 Notes on Reusability

More ZIP parsers can be added for testing. Refer to `01-infozip` as an example to configure the Docker image.

ZIPDIFF may also be adapted to test other archive formats, if the ZIP-level mutations are replaced with other format-specific ones.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Tady: A Neural Disassembler without Structural Constraint Violations

Siliang Qin[1,2], Fengrui Yang[3], Hao Wang[3], Bolun Zhang[1,2], Zeyu Gao [3], Chao Zhang [3], Kai Chen [1,2]

[1] *Institute of Information Engineering, Chinese Academy of Sciences, China*

[2] *School of Cyber Security, University of Chinese Academy of Sciences, China*

[3] *Tsinghua University, China*

{qinsiliang, zhangbolun, chenkai}@iie.ac.cn

{yangfr23, hao-wang20, gaozy22}@mails.tsinghua.edu.cn, chaoz@tsinghua.edu.cn

A Artifact Appendix

A.1 Abstract

The artifact, Tady, is a novel neural network-based disassembler designed to address structural constraint violations commonly found in the output of existing disassemblers. It consists of a neural model with a hybrid local-global attention mechanism to better understand code context and a post-processing algorithm that uses a Post-Dominator Tree (PDT) to enforce structural consistency. The artifact includes the source code for the Tady model, the PDT-based error detection and pruning algorithms, evaluation scripts, and the datasets used in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All analyzed binaries are sourced from public academic datasets. There are no destructive steps, and the artifact does not require elevated privileges or disabling of security mechanisms. The research follows the principles of the Menlo Report and USENIX Security's ethical guidelines. The dual-use potential is acknowledged, but the focus is on advancing defensive binary analysis capabilities.

A.2.2 How to access

The artifact is publicly available under an open-source license. A snapshot of the evaluated version is permanently archived on Zenodo with the DOI: 10.5281/zenodo.15541311, which can be accessed from https://doi.org/10.5281/zenodo.15541311. The latest version of the source code is maintained and updated at the following GitHub repository: https://github.com/5c4lar/tady, where the commit *014cd1d* is used for artifact evaluation.

A.2.3 Hardware dependencies

The experiments were conducted on a machine with the following specifications:

- **CPU:** Intel Core i9-12900K
- **GPU:** NVIDIA RTX A6000 Ada generation (A GPU is required for model training and inference)
- **RAM:** 64GB
- **Storage:** 1TB NVMe SSD

A machine with similar or better specifications is recommended for reproducing the results in a timely manner. Since the intermediate results generated for the datasets can get large, we recommend at least 256GB storage are available before reproducing the results. Machine with NVIDIA GPUs other then this specific model should also work smoothly.

A.2.4 Software dependencies

The artifact can be run on host or within a Docker container.

- **Operating System:** Ubuntu 24.04.
- **Core Dependencies:**
  - C++ toolchain (build-essential)
  - LLVM 18 (llvm-dev)
  - Boost Graph Library (libboost-graph-dev)
  - Python 3 development headers (libpython3-dev)
- **Python Environment:** Managed via *uv*. Key packages include Flax, JAX, TensorFlow, NumPy.
- **Baselines:** Reproducing baseline results requires installing IDA Pro (v9.1), Ghidra (v11.3.2), ddisasm (v1.9.0), DeepDi, and XDA, each with its own specific environment (e.g., conda for XDA).

- **Docker:** A Dockerfile is provided to build a container with all necessary dependencies for Tady.

The README.md file provides detailed instructions for setting up both host and Docker environments. The artifact include the disassembly results of the baselines, which can be used if the baseline software itself is not accessible.

A.2.5 Benchmarks

The evaluation uses several public and custom datasets, all of which are provided and can be generated with the artifact's scripts. The primary datasets are:

- **Pangine:** Binaries compiled with various compilers and optimizations, with labels extracted during compilation.
- **Assemblage:** Open-source Windows programs.
- **x86-sok:** Linux binaries, including both common utils and other more complex applications.
- **rw:** Real-World dataset created following the DeepDi protocol using x86-sok's toolchain.
- **quarks:** Binaries obfuscated with Tigress and OLLVM under various obfuscation settings.
- **obf-benchmark:** 11 binaries obfuscated with binobf.

The artifact provides scripts to download, preprocess, and format these datasets. Pre-processed datasets are also available for download to expedite evaluation.

A.3 Set-up

A.3.1 Installation

The following steps describe the installation process using the provided Docker environment.

1. Use the `tady-main.zip` from zenodo of clone the artifact repository:
   `git clone https://github.com/5c4lar/tady.git && cd tady`
2. Build the Docker image:
   `docker build -t tady -f docker/Dockerfile .`
3. Download and extract the pre-processed datasets, baselines' results and trained models. The artifact provides the datasets with *bin.tar.gz* and *gt_npz.tar.gz*. This avoids the lengthy process of generating datasets from scratch. Baselines' results are given in *eval_strip_baselines.tar.gz*. The models are provided in *models.tar.gz*.

```
tar -xzvf bin.tar.gz -C data
tar -xzvf gt_npz.tar.gz -C data
tar -xzvf eval_strip_baselines.tar.gz \
-C data
tar -xzvf models.tar.gz
```

At this point, the environment is ready for evaluation.

A.3.2 Basic Test

To verify the installation, run a simple disassembly test on a the VMProtect obfuscated binary in appendix. This test will use the Tady model to disassemble a file and then run the pruning algorithm on the output, For evaluation purpose, the ground-truth is used to evaluate the accuracy after pruning.

1. Start the tensorflow serving on the host:

```
docker run --rm --gpus device=0 \
-p 8500:8500 \
-v  $PWD/models/tf_models:/models -t \
--name tensorflow-serving \
tensorflow/serving:latest-gpu \
--xla_gpu_compilation_enabled=true \
--enable_batching=true \
--batching_parameters_file=\
/models/batching.conf \
--model_config_file=/models/model.conf
```

2. Enter the Tady docker container (if using Docker).
3. Run the evaluation script on a single file:

```
uv run -m tady.infer \
--path data/obf/TestApp.vmp.exe \
--model instruction_cpp_pangine_\
lite_all_64lw_64rw_16h_2l_prev000 \
--section_name .vmp0 \
--output_path \
data/obf/tady/TestApp.vmp.exe.npz \
--seq_len 569038 --batch_size 1
```

4. Run the pruning script on the result:

```
uv run -m tady.prune \
--gt data/obf/TestApp.vmp.exe.npz \
--pred data/obf/tady/TestApp.vmp.exe.npz
```

A successful run will produce a numpy file at `data/obf/tady/TestApp.vmp.exe.npz`, containing the disassembly results. The *tady.prune* script will show the precision, recall, F1 score, false positives, false in the console. The first run of the *tady.infer* will be slower than the following runs because of tensorflow-serving's warmup.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Tady's model achieves high instruction-level accuracy (Precision, Recall, F1 score), outperforming or performing competitively with state-of-the-art disassemblers on standard and obfuscated benchmarks. This is proven by experiment (E1), with results in Table 3 of the paper.

(C2): Tady's post-processing pruning algorithm effectively eliminates all detected structural constraint violations from the model's output, improving consistency while maintaining or improving F1 score. This is proven by experimen (E1), with results in Tables 3.

(C3): Existing disassemblers, including rule-based tools and neural models, frequently produce outputs with structural constraint violations, and these violations are exacerbated by obfuscation. Tady's algorithm can systematically detect these errors. This is proven by experiment (E2), with results in Tables 1 and 2 of the paper.

(C4): The entire Tady workflow is highly efficient, with time usage scaling linearly with the size of the input binary. This is proven by experiment (E3), with results in Figures 8, 9, 10, and 11.

(C5): The novel components of Tady's model, specifically the Masked Sliding Window Attention (MSWA) and Message Passing (MP), are critical to its performance. This is proven with the results of experiments (E1, E2), with results in Tables 4 and 5.

A.4.2 Experiments

Before measuring the results, follow the instructions in the Eval.Tady section in the README to generate the disassembly of Tady. The variants in the ablation study correspond to the combinations of configurations: `attention=lite,sliding connections=none,all`. Sample outputs can be found in `artifacts.tar.gz`.

(E1): Pruning and Accuracy Evaluation [30 minutes]: This experiment runs the pruning algorithm on all disassembly results and evaluates the F1 score, precision, and recall before and after pruning.
Preparation: Ensure all the disassembly results are available either by downloading from the artifacts or reproduced according to the README.
Execution: Run the following scripts.

```
uv run scripts/experiments/prune.py -m \
test_dataset=obf-benchmark,rw,x86_dataset,\
pangine,quarks,assemblage \
process=24 num_samples=1000
uv run scripts/experiments/collect_stat.py
uv run scripts/experiments/prune.py -m \
test_dataset=obf-benchmark,rw,x86_dataset,\
pangine,quarks,assemblage process=24 \
models="[gt]"
```

Results: `data/prune/all_prune_result.json`. The data in this file can be used to reproduce the accuracy results in Table 3 and Table 5, validating claims C1, C2 and C5.

(E2): Detecting Constraint Violations [30 minutes]: This experiment runs Tady's error detection algorithm over the outputs of various disassemblers (Tady, baselines) and dataset labels across all benchmarks.
Preparation: Ensure E1 has been completed and the intermediate results in `data/prune` are available. The errors are already detected in our pruning scripts.
Execution: Run the error statistics collection script. This script aggregate the error detection results.

```
uv run scripts/experiments/error\_stat.py
```

Results: `data/error_stat.json`, which contains the aggregated data used to generate Tables 1, 2, and 4 in the paper. This validates claim C3, C5.

(E3): Efficiency Benchmark [2 hours]: This experiment measures the runtime and memory usage of Tady. The time is mainly spent on slow baselines on large binaries.
Preparation: Environments to run the disassemblers.
Execution: Please follow the Efficiency section in README for instructions about this evaluation.
Results: Data of Figures 8, 9, 10, and 11, validating claim C4. The figures can be reproduced with the script `scripts/ablation/generate_uniform_plots.py` with the provided data in `argifacts.tar.gz`.

A.5 Notes on Reusability

The artifact is designed for reusability. Beyond reproducing the paper's results, it can be extended in several ways:

- **Applying to New Binaries:** The Tady inference pipeline (*tady.infer*) can be used to disassemble new, unseen x86/x86-64 binaries. The README provides an example of this in the VMProtect evaluation.
- **Training on New Datasets:** The training pipeline (*scripts/experiments/train.py*) can be adapted to train the Tady model on new, custom-labeled disassembly datasets. The data preprocessing scripts show how to convert ground truth into the required format.
- **Applying to other disassemblers** The error detection and pruning algorithms can be applied to other disassembly results. Examples on how to use the code can be found in the evaluation scripts.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models

Ahmed Lekssays[1*], *Hamza Mouhcine*[1*], *Khang Tran*[2], *Ting Yu*[3], *Issa Khalil*[1]
[1]Qatar Computing Research Institute, [2]New Jersey Institute of Technology,
[3]Mohamed bin Zayed University of Artificial Intelligence
{alekssays, hmouhcine, ikhalil}@hbku.edu.qa,
kt36@njit.edu, ting.yu@mbzuai.ac.ae
* *Joint first authors with equal contribution*

A Artifact Appendix

A.1 Abstract

LLMxCPG is a novel framework that integrates Code Property Graphs (CPG) with Large Language Models (LLM) for robust vulnerability detection. Our CPG-based slice construction technique significantly reduces code size, ranging from 67.84% to 90.93%, while preserving vulnerability-relevant context. This approach enables a more concise and accurate representation of code snippets, facilitating the analysis of larger code segments, including entire projects. Empirical evaluation demonstrates LLMxCPG's effectiveness on verified datasets, achieving 15-40% improvements in F1-score over state-of-the-art baselines. Furthermore, LLMxCPG maintains high performance across both function-level and multi-function codebases and exhibits robust detection efficacy under various syntactic code modifications.

A.2 Description & Requirements

This section lists all the information necessary to recreate the experimental setup used to run the artifact. Where it applies, the minimal hardware and software requirements to run the artifact are specified.

A.2.1 Security, privacy, and ethical concerns

Our vulnerability detection research adheres to responsible disclosure protocols and established security research guidelines. We carefully balance the benefits of identifying security weaknesses against potential risks, and all discovered vulnerabilities are reported through appropriate channels, allowing sufficient time for patches before public disclosure. We maintain strict confidentiality throughout the research process to ensure our methods do not compromise system integrity or user privacy.

A.2.2 How to access

The source code, fine-tuned models, and testing datasets used in this study are publicly available to the community to foster research in this field. They can be accessed at:

- GitHub: https://github.com/qcri/1lmxcpg
- Zenodo: https://zenodo.org/records/15614095

A.2.3 Hardware dependencies

The fine-tuning of the models was performed on an NVIDIA A100-80GB GPU. While the fine-tuned model for query generation (LLMxCPG-Q) was limited to a 32K token context due to computational constraints, the base model supports a 128K token context, with versions supporting up to 1M tokens being available if substantial computational resources are provided. For optimal performance in evaluation, access to a GPU, ideally an NVIDIA A100 with at least 80GB of memory, is recommended.

A.2.4 Software dependencies

The artifact requires the following software components:

- Operating System: Ubuntu 22.04
- Joern: An open-source static analysis tool. A cluster of Joern servers deployed using Docker containers is used for parallel processing.
- LLaMA-Factory framework
- vLLM inference library
- Python 3.x and associated packages listed in the 'requirements.txt' file within the repository.

A.2.5 Benchmarks

The evaluation utilized several datasets:

- **Training Datasets: (both for query generation and vulnerability detection)**
  - FormAI-v2
  - PrimeVul
- **Generalizability/Test Datasets:**
  - SVEN
  - Repos Vul
  - 2025 Post-Knowledge-Cutoff (PKCO-25) CVEs dataset (compiled from CVEs published in 2025)

A.3 Set-up

This section includes all the installation and configuration steps required to prepare the environment for the evaluation of the artifact.

A.3.1 Installation

To install and set up the artifact, follow these steps:

1. Clone the LLMxCPG repository from GitHub: `git clone https://github.com/qcri/llmxcpg.git`
2. Navigate to the cloned directory: `cd llmxcpg`
3. Install the required Python dependencies. A 'requirements.txt' file is provided in 'requirements.txt': `pip install -r requirements.txt`
4. Set up Joern: The paper mentions deploying Joern servers using Docker containers for parallel processing. Follow the official Joern documentation for installation and setup within a Docker environment.

A.3.2 Basic Test

To run a simple functionality test of the LLMxCPG system (specifically the detection model), you can execute the 'detect_inference.py' script with a sample code snippet.

1. Run the inference script: `python inference/detect_inference.py {dataset}` where *dataset* is one of the following: *primevul*, *formai*, *sven*, *reposvul*, and *pkco*.
2. Expected output: The model should classify the code as either "VULNERABLE" or "SAFE". For a vulnerable input, the expected output is "VULNERABLE". Then, the Accuracy, Precision, Recall, and F1-Scores are calculated.

A.4 Evaluation workflow

This section includes all the operational steps and experiments which must be performed to evaluate if your artifact is functional and to validate your paper's key results and claims.

A.4.1 Major Claims

(C1): *LLMxCPG achieves high performance on function-level vulnerability detection across diverse datasets.* This is proven by the experiments (E1) and (E2) described in Section 4.3.2 "Function-level Vulnerability Detection" and Section 4.4.1 "Function-level Vulnerability Detection" (Generalizability subsection) , whose results are illustrated/reported in Table 3 and Table 5. Specifically, LLMxCPG achieves up to 0.8146 Accuracy and 0.8075 F1-score on the FormAI dataset , and a remarkable 20% improvement in accuracy over competing approaches on the SVEN dataset.

(C2): *LLMxCPG demonstrates robust generalization capabilities on project-level codebases and novel vulnerabilities that emerged after its knowledge cutoff.* This is proven by the experiment (E3) described in Section 4.4.2 "Project-level Vulnerability Detection" , whose results are illustrated/reported in Table 7. LLMxCPG achieved an F1-score of 0.610 and Accuracy of 0.634 on the Repos Vul dataset, and an F1-score of 0.617 and Accuracy of 0.600 on the 2025 Post-Knowledge-Cutoff (PKCO-25) dataset.

A.4.2 Experiments

(E1): *Function-level Vulnerability Detection Performance on Trained Datasets [30 human-minutes + 1 compute-hour]:*
How to: This experiment evaluates LLMxCPG's performance on function-level code snippets using the FormAI and PrimeVul datasets, which were part of the training data.
Preparation: Ensure the LLMxCPG environment is set up and the fine-tuned LLMxCPG-D model (based on QwQ-32B-Preview) is loaded. Use the test splits of the FormAI and PrimeVul datasets.
Execution: Run the LLMxCPG inference pipeline on the test sets of FormAI and PrimeVul by running `python inference/detect_inference.py primevul` and `python inference/detect_inference.py formai`.
Results: The expected outcome is high performance in vulnerability detection, as shown in Table 3. For FormAI, expected Accuracy is 0.8146 and F1-score is 0.8075. For PrimeVul, expected Accuracy is 0.7250 and F1-score is 0.6206.

(E2): *Generalizability to Unseen Function-level Datasets [30 human-minutes + 1 compute-hour]:*

How to: This experiment assesses the generalizability of LLMxCPG by evaluating its performance on the SVEN dataset, which was not part of the training data.
Preparation: Ensure the LLMxCPG environment is set up and the fine-tuned LLMxCPG-D model is loaded. Use the SVEN dataset for testing.
Execution: Run the LLMxCPG inference pipeline on the SVEN dataset: `python inference/detect_inference.py sven`
Results: The expected outcome is that LLMxCPG significantly outperforms the baselines, achieving a substantial improvement in accuracy. As shown in Table 5, LLMxCPG is expected to achieve an Accuracy of 0.6020 and an F1-score of 0.7048.

(E3): *Project-level Vulnerability Detection and Generalization to Post-Knowledge-Cutoff CVEs [45 human-minutes + 1.5 compute-hour]:*
How to: This experiment evaluates LLMxCPG's performance on complex, project-level code snippets from the Repos Vul dataset and its generalization to newly published CVEs (PKCO-25 dataset).
Preparation: Ensure the LLMxCPG environment is set up and the fine-tuned LLMxCPG-D model is loaded. Use the sampled Repos Vul dataset and the 2025 Post-Knowledge-Cutoff (PKCO-25) CVEs dataset for testing.
Execution: Run the LLMxCPG inference pipeline on both the Repos Vul and PKCO-25 datasets with `python inference/detect_inference.py reposvul` and `python inference/detect_inference.py pkco`.
Results: The expected outcomes are promising results despite the complexity of the datasets. As shown in Table 7, for Repos Vul, expected Accuracy is 0.634 and F1-score is 0.610. For PKCO-25, expected Accuracy is 0.600 and F1-score is 0.617.

A.5 Notes on Reusability

All source code and datasets used in this study are open-source, supporting reproducibility, transparency, and further research in the domain of software security. The artifact provides a valuable framework for future research by enabling the analysis of large code segments through its CPG-based slice construction technique. Additionally, LLMxCPG's ability to precisely identify and isolate security-critical changes between vulnerable and patched versions makes it particularly valuable for understanding vulnerability fixes and generating high-quality training data for vulnerability detection models. This functionality can be leveraged by other researchers to compile large, high-quality datasets for training vulnerability detection models in the future.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: X.509DoS: Exploiting and Detecting Denial-of-Service Vulnerabilities in Cryptographic Libraries using Crafted X.509 Certificates

Bing Shi[1], Wenchao Li[1], Yuchen Wang[1], Xiaolong Bai[1]*, Luyi Xing[2]*

[1]*Alibaba Group*, [2]*Indiana University Bloomington*
{*shibing.shi*, *huiren.lwc*, *tianwu.wyc*}*@alibaba-inc.com*, *bxl1989@gmail.com*, *luyixing@iu.edu*

A Artifact Appendix

This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of our artifact.

A.1 Abstract

X.509DoSTool is developed to facilitate the rapid generation of crafted certificates and the automatic detection of DoS risks in cryptographic implementations. The tool primarily provides three commands for users: the `generate` command creates crafted certificates that can be used to identify specific DoS risks discussed in our paper; the `edit` command allows modification of ASN.1 objects within a certificate without violating DER encoding rules; and the `detect` command identifies specific issues in the implementations of the cryptographic libraries mentioned in our paper. The artifact evaluation should primarily focus on these three commands. We provide the source code of the tool and the necessary test scripts to facilitate the evaluation process.

A.2 Description & Requirements

This section lists the information necessary to recreate the same experimental setup we have used to run our artifact.

A.2.1 Security, privacy, and ethical concerns

Conducting the artifact evaluation for *X.509DoSTool* does not raise any security, privacy, or ethical concerns.

A.2.2 How to access

The artifact is available at our Zenodo repository (please refer to v1.0.2 and later): https://zenodo.org/records/14726326.

*Xiaolong Bai and Luyi Xing are corresponding authors.

A.2.3 Hardware dependencies

No special hardware dependencies are required for our artifact. Our testing environment is built on a Linux server equipped with 4GB of RAM and a 4-core CPU, running Ubuntu v22.04.

A.2.4 Software dependencies

The software dependencies required to run this tool primarily include the following:

- Python v3.10.12
- pip v22.0.2
- setuptools v75.8.0 (Python package)
- psutil v6.0.0 (Python package)
- pyasn1 v0.6.0 (Python package)
- pyasn1_modules v0.4.0 (Python package)
- pycryptodome v3.20.0 (Python package)
- OpenSSL v3.0.2

The versions used during testing should match or closely align with those mentioned above, as significant version discrepancies may introduce potential untested issues.

Additionally, before evaluators attempt to use the tool to detect issues in a specific library, the corresponding version of the library (as listed in Table 3 and Table 4 of our paper) needs to be installed first. The libraries and their version information involved include:

- OpenSSL v3.0.0, v1.0.2
- Botan v1.11.21, v1.11.26, v3.2.0, v3.4.0
- Bouncy Castle v1.70, v1.77
- Crypto++ v5.6.4, v8.9
- GnuTLS v3.7.10, v3.7.11
- phpseclib v3.0.18, v3.0.33, v3.0.35

A.2.5 Benchmarks

None.

A.3 Set-up

This section includes the installation and configuration steps required to prepare the environment to be used for the evaluation of our artifact.

A.3.1 Installation

To install, run `pip install .` in the root directory of the repository. This will package the necessary files and create a command named `x509dostool`.

Upon installation, the command is readily available for the root user. However, for non-root users, it is necessary to first run `export PATH=$PATH:~/.local/bin` before the command can be used.

A.3.2 Basic Test

To execute, run `x509dostool`. If the installation is successful, it will display the tool's version number along with a help page containing the three basic subcommands: `generate`, `edit`, and `detect`.

A.4 Evaluation workflow

This section includes the operational steps and experiments required to evaluate the functionality of our artifact and validate the key results and claims of our paper.

A.4.1 Major Claims

Our major claims include the following:

(C1): *X.509DoSTool* can be used to rapidly generate crafted certificates that facilitate the identification of specific DoS risks discussed in our paper.
(C2): *X.509DoSTool* can be used to modify ASN.1 objects involved in a certificate without violating DER encoding rules.
(C3): *X.509DoSTool* can be used to identify specific issues in the implementations of the cryptographic libraries mentioned in our paper.

A.4.2 Experiments

The corresponding experimental steps are as follows:

(E1): Test the `generate` command of `x509dostool`, and use `openssl` to examine the generated certificate.
Preparation: Enter the `test/generate/` directory.
Execution: Run `./test_generate.sh`.
Results: The script prints each executed `generate` command and checks the execution status, as well as the parsing status of the generated certificate using `openssl`. If all commands pass the aforementioned checks, the message *all tests passed* will be displayed. Additionally, running `./test_generate.sh --verbose` will output the intermediate results of the `generate` and `openssl` commands, providing evaluators with further details for evaluation.

(E2): Test the `edit` command of `x509dostool`, and use `openssl` to examine the edited certificate.
Preparation: Enter the `test/edit/` directory.
Execution: Run `./test_edit.sh`.
Results: Upon initial execution, a directory named `certs/` will be created, containing the certificates to be utilized. The script then prints each executed `edit` command and checks the execution status, as well as the parsing status of the edited certificate using `openssl`. If all commands pass the aforementioned checks, the message *all tests passed* will be displayed. Additionally, running `./test_edit.sh --verbose` will output the intermediate results of the `edit` and `openssl` commands, providing evaluators with further details for evaluation.

(E3): Test the `detect` command of `x509dostool`, and compare it with the experimental results presented in Section 7 of our paper.
Preparation: Enter the `test/detect/` directory.
Execution: Run `./test_detect.sh`.
Results: Upon initial execution, a directory named `certs/` will be created, containing the certificates to be utilized. The script then prints the cryptographic libraries present in the current environment along with their respective versions, and outputs the results of the detection of issues within these libraries. Note that the script `test_detect.sh` performs detection against the cryptographic libraries present in the evaluator's current environment. Given the tool's requirements, at least one OpenSSL library will be present in the environment. The evaluator can install the corresponding library of interest in the environment for detection.

A.5 Notes on Reusability

Note that the version numbers of the cryptographic libraries tested, as indicated in our paper, are not strictly required for reproducing the experimental results. In most cases, installing an earlier version to test as many cases as possible for detection is also feasible. However, this does not imply that any version earlier than these can be used, as some issues were introduced only after specific version updates. Additionally, if users wish to apply this tool to libraries beyond

those discussed in this paper, they can refer to the scripts in `test/scripts/` for creating corresponding scripts. Users can also specify other parameters during the execution of the `generate` command or use the `edit` command to enrich the contents of the `test/detect/certs/` directory.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: CoVault: Secure, Scalable Analytics of Personal Data

Roberta De Viti[1], Isaac Sheff[1*], Noemi Glaeser[2,4*], Baltasar Dinis[3*], Rodrigo Rodrigues[3], Bobby Bhattacharjee[4], Anwar Hithnawi[5*], Deepak Garg[1], and Peter Druschel[1]

[1]Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus
[2]Max Planck Institute for Security and Privacy (MPI-SP)
[3]Instituto Superior Técnico (ULisboa), INESC-ID
[4]University of Maryland
[5]ETH Zürich

A Artifact Appendix

A.1 Abstract

There is growing awareness that the analysis of personal data, such as individuals' mobility, financial, and health data, can provide significant societal benefits. However, liberal societies have largely refrained from such analytics, arguably due to the absence of secure platforms that can scale to billions of records while operating under a very strong threat model. We argue that a key missing piece is an architecture capable of scaling (actively-)secure multi-party computation (MPC) horizontally without weakening security. To bridge this gap, we introduce CoVault, an analytics platform that combines server-aided MPC with trusted execution environments (TEEs). This approach enables the colocation of MPC parties in a single datacenter without reducing security while scaling MPC horizontally up to the datacenter's available resources.

Our artifact includes the source code of our prototype implementation of CoVault, along with the scripts to execute the code and produce all the major evaluation results that are presented in the technical paper. Additionally, we provide Python scripts for visualizing the results.

A.2 Description & Requirements

We outline the hardware and software requirements for running our artifact in § A.2.3 and § A.2.4, respectively. The `README.md` file in our artifact includes step-by-step instructions to help users configure their own machines for running CoVault provided they meet the requirements. We refer to this scenario as the "generic setup", which is relevant to a generic user.

*Affiliation at time work was done.

To facilitate artifact evaluation, we provide USENIX Security evaluators with SSH access to a set of pre-configured virtual machines (VMs) on Google Cloud Compute Engine (GCE). These VMs satisfy all hardware and software dependencies, and are set up for intercommunication, as well as mutual authentication. We refer to this scenario as the "GCE setup", which is relevant to the USENIX Security evaluators.

The remainder of this section and § A.3 provide further details on hardware and software dependencies (§ A.2.3, § A.2.4) and installation steps (§ A.3), However, evaluators with SSH access to the GCE setup may skip these and proceed directly to § A.4.

A.2.1 Security, privacy, and ethical concerns

The artifact poses no security, privacy, or ethical concerns. It does not compromise system integrity, and all datasets are synthetic, hence devoid of any sensitive personal data. (In fact, CoVault uses data-oblivious computation protocols, whose performance characteristics are independent of the actual data.)

A.2.2 How to access

The artifact is available at https://zenodo.org/records/14736568. During the evaluation phase, evaluators will receive the SSH keys to access our pre-configured GCE setup, as described above.

A.2.3 Hardware dependencies

Generic setup. Our CoVault prototype runs on an interconnected set of machines (physical or virtual), where half support one second-generation TEE type (e.g., Intel TDX) and the other half support a different second-generation TEE type

(e.g., AMD SEV-SNP). Second-generation TEEs encapsulate entire VMs, unlike first-generation TEEs (e.g., Intel SGX). The minimum requirements to run CoVault is one machine of each TEE type, each with at least 2 CPU cores.

To run all the scripts reproducing the experiments in the paper, each machine must have at least 4 CPU cores, 32 GB RAM, and 50 GB of free disk space. Insufficient disk space may result in truncated output logs (including latency results) and the impossibility to run tests with the AG2PC protocol, which requires to store large precompiled MPC circuits.

The results in the evaluation section of the paper were obtained in a setup with at least 15Gbps interconnect bandwidth among the machines (Google gVNIC on GCE). While CoVault can run with lower bandwidth, this may cause network bottlenecks, increased latencies, and, in the case of our prototype, timeouts.

An experiment in the paper measures the cost of TEE overhead in CoVaultṪo this end, it compares CoVault to an otherwise identical setup without the TEEs. To run this experiment, the user must either (i) have the ability to enable and disable TEE encapsulation on the same machine pair, or (ii) have access to two identical (physical or virtual) machines pairs – one with the two active TEEs of different types and one without TEEs.

Finally, the horizontal scaling experiments in the paper use varying VM and core configurations: (i) 2 VMs with at least 2 cores each, (ii) 2 VMs with at least 4 cores each, (iii) 4 VMs with at least 2 cores each, (iv) A total of 22 cores per TEE type, preferably distributed across multiple VMs. The `README.md` file specifies the required configuration for each experiment. Reproducing a scaling experiment requires its corresponding resources.

GCE setup. The GCE setup meets all of the above requirements. In particular, it is preconfigured with a sufficient number of Cloud VMs encapsulated into either Intel TDX or AMD SEV-SNP TEEs to run all experiments. Furthermore, it includes additional VMs with identical hardware and software configurations but without active TEEs for measuring TEE overhead. The Intel VMs are c3-standard-8 and the AMD VMs are n2d-standard-8; they are deployed in the us-central1-a zone and use the gVNIC (15Gbps). The `README.md` file specifies which set of GCE VMs to use for each experiment.

A.2.4 Software dependencies

Generic setup. Our codebase has been tested on Ubuntu 22.04 LTS and Debian 12. While it is expected to work on other Unix-based environments, dependency names may vary from those automatically installed by our setup scripts. The artifact requires C++ build tools, Python3, and pip3 for installing Python packages. All other dependencies are installed by `setup.sh`, as detailed in the `README.md` file.

GCE setup. The guest OS on all GCE VMs is Ubuntu 22.04 LTS, and all the software dependencies have already been installed.

A.2.5 Benchmarks

As noted in § A.2.1, CoVault runtime cost is a function of the input dataset size and fixed integer width, but it is independent of the actual data. Therefore, our experiments use only synthetic datasets, which are generated by our code.

Generic setup. The `README.md` file provides step-by-step instructions for generating these datasets and populating a Redis database.

GCE setup. On the GCE setup, this data generation process has already been completed.

A.3 Set-up

Generic setup. Once all hardware and software dependencies are met, the user can obtain the artifact from the Zenodo URL in § A.2.2. The `README.md` file provides instructions for installing dependencies, setting up the CoVault prototype, writing `CONFIG.me` files, and troubleshooting common issues.

GCE setup. On the GCE setup, the artifact is already installed and available at `/home/covault`. Additionally, all required configuration files are already set up.

A.3.1 Installation

Generic setup. As mentioned in § A.2.4, our software dependencies are the standard C/C++ build tools, Python3, and pip3. Our setup scripts intentionally do not install these dependencies, as the process is OS-specific.

On Ubuntu 22.04 LTS, the required software can be installed with: (i) `sudo apt install build-essential`; (ii) `sudo apt install python3`; (iii) `sudo apt install python3-pip`. Additionally, running `sudo apt update` may be useful in some cases. (Note that our scripts should not be sensitive to specific versions of Python3, but we have only tested with Python 3.11.5.)

After downloading the artifact (§ A.2.2), users can follow the instructions in `README.md` to install CoVault and its additional dependencies.

GCE setup. On the GCE setup, CoVault is preconfigured for evaluation – no additional installation is required.

A.3.2 Basic Test

The `README.md` file provides step-by-step instructions for running basic tests like the `primitives.sh` script.

A.4 Evaluation workflow

A.4.1 Major Claims

Next, we summarize the main experimental results claimed in Sec. 6 of the CoVault technical paper. These claims, (C1)–(C5) below, can be verified by running the corresponding experiments (E1)–(E5) in § A.4.2. We begin with claims related to the choice of the underlying MPC protocol and the performance of basic query primitives in CoVault:

(C1) DualEx is a compelling choice for the protocol underlying CoVault, when a 1-bit leak is acceptable. On one core pair, it runs twice as long as semi-honest garbled circuits (GCs) by design and it is about 10x faster than the fully maliciously-secure protocol, AG2PC. The overhead of TEEs in CoVault is negligible compared to MPC (e.g., under 1s in our experiment), with MPC being the main bottleneck. This claim is verified by experiment (E1) in § A.4.2 and described in Sec. 6.2, Figure 3 of the technical paper.

(C2) The runtime of linear table scans increases linearly with the input size, while sorting, sorted merge, and compaction exhibit slightly super-linear growth. Sorting is significantly more expensive than compaction (which is why our reduce trees sort only in the first stage and merge-compact in subsequent stages). This claim is established using experiment (E2) in § A.4.2 and described in Sec. 6.2, Figures 4a and 4b of the paper.

We further evaluate CoVault in the context of an epidemic analytics scenario:

(C3) Query latency scales almost inversely with the number of available core pairs (of the two TEE types). Even with only 4 core pairs (i.e., 4 cores on 2 VMs), our implementation of basic FGA queries completes in a reasonable time. For instance, queries q1 and q2 in the technical paper complete within 20min and 3h in our experiments on 4 core pairs, and their execution time improves in proportion to the number of core pairs available to our MapReduce setup. This claim is established by experiment (E3) and is described in Sec. 6.3, Figure 6 (blue lines) of the technical paper.

In addition to our empirical tests, we developed a performance model based on detailed measurements of basic units (individual mappers, reducers, and other query primitives) to extrapolate query latency at scale (Sec. 6.3.1 of the technical paper):

(C4) Our performance model correctly predicts the empirical results of q2 in small deployments, provided that the measurements of basic units are taken on the same testbed as full queries. This claim is established by experiments (E4) and (E5), and the corroboration between the extrapolated results and empirical results on our small deployment is shown in Sec. 6.3 of the technical paper, Figure 6b (yellow line).

(C5) We claim that colocation of the computing parties is key to scaling MPC-based computations. Our performance model indicates that executing epidemic analytics for a country with 80M people in a reasonable amount of time requires substantial bisection bandwidth, which is difficult to achieve across datacenters. This claim is based on results from our performance model (experiments (E4) and (E5)), as discussed in Sec. 6.3.1 of the technical paper.

A.4.2 Experiments

The artifact includes a `README.md` file detailing the steps to configure the environment, run experiments, collect results, and generate figures supporting claims C1–C5 in § A.4.1. While the GCE setup is preconfigured for immediate use, the `README.md` also provides guidance on how to configure a generic setup for running the experiments successfully.

We summarize the experiments below, linking them to the claims they support. The human hours assume access to our GCE setup. The compute hours are approximate, but we recommend planning conservatively. In fact, using a testbed on a public Cloud such as GCE introduces higher variability than an in-house controlled setup due to factors like resource contention, network fluctuations, background load from other tenants, and potential hardware allocation variations by the Cloud provider.

For this reason, although a single experiment may not take long, multiple repetitions are required to obtain stable and repeatable results, extending overall runtime. The compute hours below refer to the default number of repetitions set in the scripts. Our evaluation in Sec. 6 of the technical paper aggregated results over two months of execution.

(E1) [0.1 human-hour + 24 compute-hours per configuration]: Microbenchmarks comparing the performance of the MPC protocols used in our evaluation, and breaking down the costs of different system components (C1).

(E2) [0.1 human-hour + 24 compute-hours per configuration]: Microbenchmarks evaluating the cost of basic query primitives as a function of the input size (C2).

Evaluation in the context of an epidemic analytics scenario:

(E3) [0.1 human-hour + 24–72 compute-hours per configuration]: Direct measurement of horizontal scaling with a small number of cores for two queries described in the paper (C3).

Extrapolation of query latency to a very large number of cores and big data:

(E4) [0.1 human-hour + 24 compute-hours per configuration]: Measurements of basic execution units and latency extrapolation using the performance model (C4 and C5).

(E5) [0.1 human-hour + 0.1 compute-hours per configuration]: Execution of the performance model and analysis of results (C4 and C5).

A.5 Notes on Reusability

The `README.md` file provides guidelines for tuning various parameters in the scripts, such as the number of repetitions, whether to execute DualEx or a semi-honest protocol for comparison, the roles of generator and evaluator in a semi-honest execution, and input sizes.

Users can also modify the C++ code to adjust parameters like the size of intermediate results between mappers and reducers and attribute bit-width. While our prototype evaluates a specific set of queries, users can implement custom FGA queries by leveraging the building blocks in `src/primitives.cpp` and replicating the MapReduce execution logic from our test files.

Note that if the input size for certain experiments is significantly increased, additional RAM may be required to process the data in-memory. Otherwise, the experiments may run out of memory and terminate prematurely.

Additionally, although our prototype evaluates queries in the context of an epidemic analytics scenario, is it possible to change the database schema and the queries.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix – EvilEDR: Repurposing EDR as an Offensive Tool

Kotaiba Alachkar
Delft University of Technology

Dirk Gaastra
Independent Researcher

Eduardo Barbaro
Delft University of Technology

Michel van Eeten
Delft University of Technology

Yury Zhauniarovich
Delft University of Technology

A Artifact Appendix

A.1 Abstract

We provide artifacts to detect EvilEDR, as described in our study. These include Sigma rules in a generic format, which can be adapted for various security tools, along with platform-specific detection queries tailored to the Endpoint Detection and Response (EDR) solutions covered in this research, including Microsoft Defender for Endpoint (MDE) using Kusto Query Language (KQL), Elastic EDR using Elastic Query Language (EQL), Sophos EDR using SQL-based queries, and Trend Micro EDR using search queries.

For environments without EDR or Security Information and Event Management (SIEM) solutions, we provide PowerShell scripts that can be executed out of the box on any Windows 11 machine. These scripts detect unauthorized EDR drivers and processes directly from the local system without requiring privileged access or external dependencies.

Our artifacts offer practical and reproducible implementations, aligning with our "Open Science" commitment. This enables the scientific community to validate our findings and helps enterprises detect EvilEDR effectively. For example, MDE users can directly apply the KQL queries to monitor for unauthorized EDR activity, while PowerShell scripts provide a quick and flexible alternative for manual checks.

A.2 Description & Requirements

To recreate the same experimental setup and evaluate the artifact functionality, no specific hardware requirements are needed. The evaluation requires Windows 11 (tested on version 22H2) and trial licenses for the EDR solutions covered in our study. The objective is to test whether the provided detection rules (tailored to different EDR platforms) can successfully identify the presence of EvilEDR, an unauthorized EDR deployed and controlled by an attacker.

The process is as follows: install Windows 11, deploy one of the supported EDRs as the Enterprise EDR, and deploy a different EDR from another vendor to act as EvilEDR. Then, import the detection rule corresponding to the Enterprise EDR into its management console. These rules identify EvilEDR drivers and/or processes using vendor-specific detection queries. For environments without an EDR or SIEM, we provide PowerShell scripts that can be executed directly on the machine. These scripts detect the presence of known EDR presence and provide a viable alternative for EvilEDR detection.

It is important to note that we included Sigma rules in a generic format, which can be mapped to other detection formats as needed. It is up to the evaluators to test and adapt them based on their specific security tools and environment. Additionally, the Ludus testbed configuration file is provided to replicate our test environment; however, evaluators may verify and modify it as needed for their testing setup.

A.2.1 Security, privacy, and ethical concerns

Our artifact does not pose any security, privacy, or ethical risks to evaluators. The detection rules and scripts provided are designed solely for identifying EvilEDR activity and do not perform any destructive actions, disable security mechanisms, or alter system configurations.

The PowerShell scripts execute locally without requiring administrative privileges, and all detection is performed in a read-only manner. No persistent changes are made to the operating system, installed software, or user data.

A.2.2 How to access

Our artifact repository is hosted on the Zenodo platform and is publicly accessible through the following permanent URL: https://doi.org/10.5281/zenodo.15116409

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Our artifact requires a Windows 11 operating system for evaluation. We tested it on Windows 11 version 22H2, but any

recent version should work. Additionally, the evaluation requires access to trial licenses for the following EDR solutions, which offer publicly available trial versions: MDE[1], Elastic EDR[2], Sophos EDR[3], and Trend Micro EDR[4].

Evaluators must obtain and install these solutions independently using the links provided. Trial versions were used during testing and provided full support for running the detection rules and queries included in the artifact. No proprietary software is included in the artifact package.

A.2.5 Benchmarks

Our artifact does not require predefined datasets, models, or workloads for evaluation. Instead, the detection rules and scripts are designed to identify the presence of EvilEDR on a machine. When evaluating detection rules for a specific vendor, the corresponding EDR must be deployed as the Enterprise EDR, and the queries are used to detect the presence of EvilEDR installed from a different vendor. If evaluators are testing the PowerShell scripts, it is sufficient to have EvilEDR deployed on the system and execute the scripts, which will list any detected drivers or processes.

A.3 Set-up

Evaluators must prepare their environment by obtaining trial licenses, installing the EDR solutions, and then running the detection artifacts. Trial licenses can be obtained directly from the vendors using the links listed in the Software Dependencies section.

A.3.1 Installation

Evaluators must set up a Windows 11 test environment and deploy two different EDR solutions: one as the Enterprise EDR and another as EvilEDR (from a different vendor). After deployment, they must import the detection rule specific to the Enterprise EDR into its management console:

- **MDE:** `MDE_EvilEDR_drivers.kql` and `MDE_EvilEDR_processes.kql`
- **Elastic:** `Elastic_EvilEDR_drivers.eql` and `Elastic_EvilEDR_processes.eql`
- **Sophos:** `Sophos_EvilEDR_drivers.txt` and `Sophos_EvilEDR_processes.txt`
- **Trend Micro:** `TrendMicro_EvilEDR_drivers.txt` and `TrendMicro_EvilEDR_processes.txt`

[1] https://learn.microsoft.com/en-us/defender-endpoint/defender-endpoint-trial-user-guide

[2] https://www.elastic.co/endpoint-detection-response

[3] https://www.sophos.com/en-us/products/endpoint-antivirus/free-trial

[4] https://www.trendmicro.com/en_hk/business/products/trials.html

To reduce false positives, evaluators should modify the detection rules to exclude drivers and processes belonging to the Enterprise EDR. If testing the PowerShell scripts, enable script execution using:

```
Set-ExecutionPolicy Unrestricted -Scope Process
```

A.3.2 Basic Test

Evaluators must run the detection rule in the Enterprise EDR management console. If EvilEDR is present, the query will return a list of detected drivers or processes; if not – it should return an empty result. For PowerShell-based detection, evaluators should run *Invoke-EDRDriverChecker.ps1* and *Invoke-EDRProcessChecker.ps1*. If EvilEDR is found, the script will list detected components; otherwise, the output will show "No EvilEDR detected."

A.4 Evaluation workflow

This section outlines the steps required to evaluate the artifact and validate Section 6 (Defense) of our research. The evaluation involves setting up a controlled test environment, deploying EvilEDR, applying detection rules using the Enterprise EDR's management console, and verifying their effectiveness. It also includes testing PowerShell-based detection methods for environments without an EDR or SIEM.

A.4.1 Major Claims

(C1): *Enterprise EDR solutions can detect the presence of EvilEDR when our detection rules are applied. This is validated through experiment (E1), where the detection rules queries successfully identify EvilEDR drivers and processes across different vendor combinations. This is described in Section 6.1 (Enterprise-Level Defenses).*

(C2): *Our PowerShell scripts provide an effective alternative to identify EvilEDR in environments without an Enterprise EDR or SIEM. This is demonstrated in experiment (E2), where running `Invoke-EDRDriverChecker.ps1` and `Invoke-EDRProcessChecker.ps1` on a test machine lists detected EvilEDR drivers and processes. This is also described in Section 6.1 (Enterprise-Level Defenses).*

A.4.2 Experiments

This section outlines the experiments required to evaluate the artifact and validate Section 6 (Defense) of our paper. Each experiment describes how to configure the test environment, execute the detection methods, and interpret the results.

(E1): *Enterprise EDR Detection [3 human-hours]: Evaluates the ability of an Enterprise EDR to detect EvilEDR*

using the provided detection rules. This experiment validates (C1).

Preparation: Install Windows 11 and deploy two different EDR solutions, one as the Enterprise EDR (e.g., MDE) and another as EvilEDR (e.g., Elastic). Import the detection rules specific to the Enterprise EDR into its management console:

- **MDE:** `MDE_EvilEDR_drivers.kql`, `MDE_EvilEDR_processes.kql`
- **Elastic:** `Elastic_EvilEDR_drivers.eql`, `Elastic_EvilEDR_processes.eql`
- **Sophos:** `Sophos_EvilEDR_drivers.txt`, `Sophos_EvilEDR_processes.txt`
- **Trend Micro:** `TrendMicro_EvilEDR_drivers.txt`, `TrendMicro_EvilEDR_processes.txt`

Repeat this process with different combinations of Enterprise EDR and EvilEDR (e.g., Elastic as Enterprise EDR and Sophos as EvilEDR, Sophos as Enterprise EDR and Trend Micro as EvilEDR) to ensure cross-compatibility testing.

Execution: Run the detection rule in the Enterprise EDR's query interface. Perform this step for each EDR pairing tested.

Results: If EvilEDR is present, the query should return a list of detected drivers or processes. Otherwise, the result should be empty. Confirm detection works consistently across different EDR combinations.

(E2): *PowerShell-Based Detection [0.5 human-hours]: Evaluates the ability to detect EvilEDR in environments without an Enterprise EDR or SIEM. This experiment validates (C2).*

Preparation: Install Windows 11 and deploy an EvilEDR (no Enterprise EDR is required). Place the following scripts in a working directory: `Invoke-EDRDriverChecker.ps1`, `Invoke-EDRProcessChecker.ps1`

Execution: Open PowerShell and run both scripts using the following commands:

```
.\Invoke-EDRDriverChecker.ps1
.\Invoke-EDRProcessChecker.ps1
```

Results: If EvilEDR is present, the scripts will list detected drivers and processes, including file paths. If no unauthorized EDR is found, the output will be: `No EvilEDR detected.`

A.5 Notes on Reusability

To reuse the detection artifacts in practical scenarios, evaluators must tune the detection rules to exclude *authorized* or *enterprise-approved* EDR processes and drivers. The provided rules are designed to identify all known EDR components, including both enterprise-deployed EDRs and unauthorized EvilEDR instances. To reduce false positives, particularly in environments with multiple active EDRs, enterprises should adapt the detection rules by explicitly excluding trusted drivers and processes while keeping detections for unauthorized instances enabled. Additionally, the PowerShell-based detection scripts are lightweight, self-contained, and can be executed without elevated privileges. They can be integrated into endpoint management solutions (e.g., Microsoft Intune) to support automated scanning, alerting, and response workflows. This enables security teams to scale EvilEDR detection across large environments and automate what would otherwise be manual investigations.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: "Nothing is Unreachable: Automated Synthesis of Robust Code-Reuse Gadget Chains for Arbitrary Exploitation Primitives"

Nicolas Bailluet
Univ Rennes, Inria, CNRS, IRISA

Emmanuel Fleury
Univ Bordeaux, CNRS, LaBRI

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA

Erven Rohou
Univ Rennes, Inria, CNRS, IRISA

A Artifact Appendix

A.1 Abstract

The paper addresses the problem of automatically chaining code-reuse gadgets to exploit vulnerabilities with arbitrary exploitation primitives. While related works generally require an attacker to control the stack (e.g. via a stack-overflow), our approach adapts to arbitrary situations – involving the stack, heap or anything else. Our experiments show that, unlike related gadget chaining approaches, our proof-of-concept tool – ARCANIST – supports various situations that involve diverse exploitation primitives and layouts – where competing tools are simply not usable. Especially, we demonstrate its applicability by generating gadget chains for 10 cases of real-world vulnerabilities involving non-trivial layouts. Additionally, we show that ARCANIST can generate long and complex chains, leveraging intricate chaining techniques – e.g. unassisted stack-pivoting, jump-oriented chaining and conditional instructions.

A.2 Description & Requirements

The artifact is distributed as a set of archives. Among them, we provide `experiments.tar.gz`, which is the only one needed to reproduce the experiments. The rest of this document will exclusively refer to `experiments.tar.gz`. Other archives are provided for reference and transparency.

A.2.1 Security, privacy, and ethical concerns

No risk is taken when executing the artifact, no destructive steps are taken, and no security mechanism is disabled during execution. Moreover, experiments are containerized and are meant to be executed in the provided container to avoid polluting the evaluator's system.

A.2.2 How to access

The artifact is made available on Zenodo under the following DOI: 10.5281/zenodo.14724513.

A.2.3 Hardware dependencies

The tools run by the artifact may consume a lot of RAM and run on a lot of cores. If you don't have enough RAM, processes might get killed by the OS. If you don't have enough cores, this might affect execution times and overall performance. We recommend running the experiments on a system with at least **96GB of RAM**, and **ideally 64 CPU cores or more** (the specifications of our experimental setup). As a reference, *our CPU cores were running at 2.3GHz.*

A.2.4 Software dependencies

The artifact is meant to be run on a **Linux operating system**. The experiments are containerized to facilitate their execution, isolation, and dependencies installation. You will need **`Docker`** to run the containers and launch the experiments.

A.2.5 Benchmarks

The binaries used for our experiments are provided in the archive in two folders:

- `./cve_binaries` for the experiments of Section 8.1 to 8.3 on several CVEs.
- `./binaries` for the experiments of Section 8.4 to 8.6.

Our tool ARCANIST uses a commercial (non-free) version of Binary Ninja to lift gadgets. Therefore, to ensure reproducibility, and for other tools as well (when possible), we provide the gadget libraries already extracted – for ARCANIST in `./arcanist/gadgets`, for SGC in `./sgc/target/*/.cache`, and for Angrop in `./angrop/gadgets`.

A.3 Set-up

A.3.1 Installation

1. Install `Docker` if not already done and ensure that you can run `docker compose`.
2. Download and extract the `experiments.tar.gz` archive from the Zenodo repository (see A.2.2).
3. Inside the extracted folder, run the following command to import the containers:

```
$ ./import-containers.sh
```

A.3.2 Basic Test

Test environments are containerized and ready-to-use, just check that you can access the containers by running:

```
$ docker compose run <TOOL>
```

with `<TOOL>` being `angrop`, `ropium`, `sgc` or `arcanist`.

In each container, a basic functionality test can be run with:

```
$ ./functionality_test.sh
```

In the end, it should print whether the basic functionality test passed or failed.

A.4 Evaluation workflow

Note that, since ARCANIST launches multiple instances in parallel, the generated chains may sometimes be different from the ones given in the paper (an instance can succeed before another, and randomness is involved in the underlying solver).

A.4.1 Major Claims

(C1): ARCANIST can generate gadget chains for situations where existing works are just unusable. This is proven by the experiment (E1) detailed in Section 8.1 of the paper, whose results are given in Table 1.

(C2): ARCANIST can generate long chains and leverage complex chaining techniques without explicit requests – such as jump-oriented chaining, unassisted stack-pivoting, use of conditional instructions and use of additional buffers. This is proven by the experiments (E2) and (E3), described in Sections 8.2 and 8.3 of the paper, whose results are illustrated in Listing 2, 3 and 4.

(C3): Even in situations where existing works are usable, ARCANIST proves to be more flexible and versatile by being able to reach more attack goals and succeeding more often than its competitors. This is proven by the experiment (E4), detailed in Section 8.4 of the paper, whose results are given in Table 2.

A.4.2 Experiments

Items (E1) to (E4) are the most important ones, dedicated to our major claims. Regarding items (E5) to (E7), they are dedicated to – less significant – time-consuming performance evaluations.

(E1): *[CVEs (Table 1)] [5 human-minutes + 2 compute-hours]:* In this experiment, we ask ARCANIST to generate gadget chains for real-world CVE situations where other tools are unusable. We expect ARCANIST to succeed in generating chains in almost all situations – except one, see Table 1 in the paper.

Execution: Firstly, enter the `arcanist` container:

```
$ docker compose run arcanist
```

Then, run the following command to start the experiment:

```
$ ./run_table1_experiments.sh
```

Results: Once done, you can print the results with:

```
python print_table1_results.py
```

This will display the results in the same order as in Table 1 in the paper.

(E2): *[Complex Chains (Listing 2 & 3)] [5 human-minutes + 30 compute-minutes]:* In this experiment, we show the complexity of ARCANIST-generated chains through two examples. We expect ARCANIST to generate non-trivial chains showing: jump-oriented chaining, use of conditional instructions and stack-pivoting. The generated chains should be similar to the paper's chains, but you might observe variations.

Execution: Firstly, enter the `arcanist` container, then, run the following commands separately:

```
$ ./run_listing2_synthesis.sh
$ ./run_listing3_synthesis.sh
```

Results: For each command, the generated chain should be printed in the end, once done.

(E3): *[OPTEE Chain (Listing 4)] [5 human-minutes + 5 compute-minutes]:* In this experiment, we show the ability of ARCANIST to generate a stack-pivoting chains for our OPTEE CVE case-study. The generated chain should be similar to the paper's chain, but you might observe variations.

Execution: Firstly, enter the `arcanist` container, then, run the following command:

```
$ ./run_listing4_synthesis.sh
```

Results: Once done, the generated chain should be printed in the end.

(E4): *[Tools Comparison] [10 human-minutes + 15 compute-hours + 90GB-disk]:* This experiment compares ARCANIST with other tools in situations where its competitors are usable. We expect ARCANIST to always succeed, while other tools should fail in some situations.

Execution: In each tool's container, run the following command:

```
$ ./run_table2_experiments.sh
```

Results: The results for each tool can be printed by running the following command in the tool's container:

```
$ python print_table2_results.py
```

This will print the results in the same layout as the tool's associated column from Table 2 in the paper.

(E5): *[Memory Writes Benchmark] [10 human-minutes + 40 compute-hours]:* This experiment evaluates the performance of ARCANIST regarding the `mem-write-ratio` parameter. We expect to observe a gap in average execution times between ratios of 0% and 5% (most important observation). The average execution time is also expected to increase as the ratio increases from 5% to 25% – the paper shows a linear evolution, but depending on the experimental setup, the linear evolution might not be as clean as Figure 7 in the paper.
Execution: Enter the `arcanist` container, then, run the following command:

```
$ ./run_figure7_experiments.sh
```

Results: You can plot the figure by running:

```
$ ./plot_figure7.sh
```

This will generate a `pdf` at `./arcanist/synthesis_results/memory_writes_benchmark/avg_time_mem_writes.pdf`, that you can open outside the container.

(E6): *[Sample Size vs Jobs Benchmark (libc)] [10 human-minutes + 5 compute-days]:* This experiment evaluates the performance of ARCANIST regarding the `jobs` and `size` parameters, for a stack-based scenario in libc. We expect the best execution times to be achieved with samples of 100 gadgets (most important observation). We also expect to see a color gradient similar to the one given in Figure 8 in the paper.
Execution: Enter the `arcanist` container, then, run the following command:

```
$ ./run_figure8_experiments.sh
```

Results: You can plot the figure by running:

```
$ ./plot_figure8.sh
```

This will generate three `pdf` in `./arcanist/synthesis_results/lib_size_vs_process_count_evaluation/libc.so.6/`, named `average_time.pdf`, `success_ratio.pdf` and `average_chain_length.pdf`, which correspond to the three heatmaps in Figure 8. You can open them outside the container.

(E7): *[Sample Size vs Jobs Benchmark (dnsmasq)] [10 human-minutes + 7 compute-days]:* This experiment evaluates the performance of ARCANIST regarding the `jobs` and `size` parameters, for a heap-based scenario in dnsmasq. We expect the best execution times to be achieved with larger samples than (E6), and execution times should be one order of magnitude higher than (E6) (most important observation). We also expect to see a color gradient similar to the one given in Figure 9 in the paper, especially there should be a lot of black squares (indicating failures/timeouts).
Execution: Enter the `arcanist` container, then, run the following command:

```
$ ./run_figure9_experiments.sh
```

Results: You can plot the figure by running:

```
$ ./plot_figure9.sh
```

This will generate three `pdf` in `./arcanist/synthesis_results/lib_size_vs_process_count_evaluation/dnsmasq/`, named `average_time.pdf`, `success_ratio.pdf` and `average_chain_length.pdf`, which correspond to the heatmaps in Figure 9. You can open them outside the container.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: From Constraints to Cracks: Constraint Semantic Inconsistencies as Vulnerability Beacons for Embedded Systems

Jiaxu Zhao[1,2,3,4], Yuekang Li[5], Yanyan Zou[1,2,3,4]✉, Yang Xiao[1,2,3,4], Naijia Jiang[1,2,3,4]
Yeting Li[1,2,3,4], Nanyu Zhong[1,2,3,4], Bingwei Peng[1,2,3,4], Kunpeng Jian[1,2,3,4], Wei Huo[1,2,3,4]✉
[1]Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, China
[4]Beijing Key Laboratory of Network Security and Protection Technology, China
[5]University of New South Wales, Australia

A Artifact Appendix

A.1 Abstract

This artifact presents NÜWA, a novel static analysis technique that leverages constraint semantic inconsistencies to detect vulnerabilities in embedded systems. NÜWA achieves scalable and precise vulnerability discovery by addressing the challenges of identifying constraint semantics across diverse implementations and accurately extracting them. We implemented NÜWA and evaluated it using known vulnerability datasets, including 31 vulnerabilities from 13 vendors, and compared its performance to five state-of-the-art (SOTA) tools.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

NÜWA is designed for discovering vulnerabilities in embedded firmware. It does not pose any direct risk to the evaluator's system security, data privacy, or other ethical concerns during its execution. NÜWA operates on firmware images in an offline manner and does not require network access or interaction with live systems. Moreover, NÜWA does not disable any system security mechanisms nor perform any destructive operations by default.

As NÜWA is open-sourced, it is technically possible for users to discover real-world vulnerabilities. However, NÜWA is released strictly for academic research purposes. We strongly encourage that any vulnerabilities discovered using the tool be reported in accordance with responsible disclosure practices, as outlined in Section 4.5 of our paper, to help protect the broader Internet ecosystem.

✉Corresponding Authors.

A.2.2 How to access

The source code necessary to evaluate NÜWA, along with the dataset used in our paper and the Python virtual environment required to run the tool, is available in our archived Zenodo repository at: https://doi.org/10.5281/zenodo.15605329. This DOI provides a stable and citable reference to the evaluated version of the artifact, as required for the Artifacts Available badge.

A.2.3 Hardware dependencies

Our evaluation of NÜWA was conducted on a host machine with a 28-core Intel Xeon processor and 256 GB of RAM running Ubuntu 22.04. However, NÜWA does not require any specific hardware features and can run on any Linux-based operating system. No specialized hardware (e.g., GPU, FPGA, or hardware counters) is needed. Higher-end configurations may improve performance, but they are not mandatory for functionality. We recommend at least 8 CPU cores and 32 GB of RAM for smooth execution and reasonable analysis time.

NÜWA can also be executed within a virtual machine environment and does not require root privileges for installation or execution.

A.2.4 Software dependencies

To evaluate NÜWA, the most critical requirement is the installation of IDA Pro (version 9.0 or 9.1) with full decompiler plugin support. Since IDA Pro is a commercial reverse engineering software, we are unable to provide a pre-configured

environment containing it. Evaluators must obtain a valid license and install the software independently.

In addition, NÜWA was evaluated using Python version 3.10.12, and we strongly recommend using the same version to ensure compatibility and reproducibility.

A.2.5 Benchmarks

We used two datasets to evaluate NÜWA. The first is a known-vulnerability dataset, included in the vuln_dataset folder, which consists of firmware samples containing vulnerabilities with publicly available ground truth. The second is a firmware dataset used for discovering previously unknown vulnerabilities, included in the firmware_dataset folder. Due to ongoing coordination with vendors and responsible disclosure processes, only a subset of the firmware images in this dataset can be made publicly available at this time.

A.3 Set-up

A.3.1 Installation

As described in Section A.2.4, to download and install dependencies as well as the main artifact, you only need to install IDA Pro (version 9.0 or 9.1) and Python (version 3.10.12 is recommended).

Then, follow the instructions provided in our Zenodo repository to configure the Python virtual environment.

Or you can use NÜWA in the Docker provided in our Zenodo repository, which only you need to install IDA Pro and configure IDALib.

A.3.2 Basic Test

- Modify 104 lines of code in main.py. Change *for todo_item in binary_todo* to "*for todo_item in binary_todo1*"
- Enter the known vulnerability folder defined in *binary_todo1*
- Run main.py in the python virtual environment.
- In the known vulnerability folder defined by *binary_todo1*, there are the outputs after firmware unpackaging, the front-end source identification output (front folder), the front-end constraint analysis results (frontend.log), and the back-end constraint analysis results (backend.log).
- In the back-end analysis results, the function summary analysis results, slicing results, constraint extraction results, and semantic inconsistencies results between back-end explicit and desired constraints are listed in detail.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): NÜWA can detect 28 known vulnerabilities and 12 additional known vulnerabilities. This is proven by the experiment E1.

(C2): The constraint extraction and constraint semantic inconsistencies results are proven by the experiment E2.

A.4.2 Experiments

(E1): *[Known Vulnerabilities Detection] [5-10 compute-hour + 80GB disk]*

Execution: Run main.py in the python virtual environment.

Results: The analysis result is in each vulnerability folder. The vulnerability detection result need to be confirmed manually.

(E2): *[Constraint Analysis] [12 human-hour]*

How to: Manually analyse the result file: frontend.log and backend.log. Among them, the semantic inconsistency between the back-end explicit constraint and the desired constraint has been automatically analysed in backend.log, while the semantic inconsistency between the front-end and back-end explicit constraints requires manual analysis of the display constraints in frontend.log and backend.log.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Exposing and Circumventing SNI-based QUIC Censorship of the Great Firewall of China

Ali Zohaib*
University of Massachusetts Amherst

Qiang Zao*
GFW Report

Jackson Sippe
University of Colorado Boulder

Abdulrahman Alaraj
University of Colorado Boulder

Amir Houmansadr
University of Massachusetts Amherst

Zakir Durumeric
Stanford University

Eric Wustrow
University of Colorado Boulder

A Artifact Appendix

A.1 Abstract

Our paper presents the first study of China's new capability to censor QUIC connections based on the Server Name Indication (SNI) field. We find that China has started decrypting QUIC Initial packets at scale, employing unique filtering rules and a distinct blocklist different from its other censorship mechanisms. We measure the blocked domains, reverse-engineer the filtering rules (e.g., filtering connections where source port > destination port), and demonstrate how modest QUIC traffic surges can overwhelm the system and reduce the effectiveness of its blocking. Furthermore, we show how the GFW's QUIC blocking mechanism can be exploited to block UDP connections between arbitrary hosts.

This artifact provides the code and data from our experiments. It is structured to make all figures and results in the paper reproducible and to enable independent verification of our claims about the GFW's QUIC-SNI blocking.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The tools used in this artifact do not pose any risk to the evaluators' machines or data privacy. All measurements performed using this artifact are non-destructive and are for verification purposes only.

A.2.2 How to access

The repository containing the code and data can be cloned from GitHub: `https://github.com/gfw-report/usenixsecurity25-quic-sni`. Archived versions of the artifact are available under the DOI: 10.5281/zenodo.15606165.

*Ali Zohaib and Qiang Zao contributed equally to this work.

A.2.3 Hardware dependencies

For the AE reviewers, we have provisioned two Virtual Private Servers (VPSes) for experiments. Both servers are configured with a single CPU core and 2GB of RAM. The first is hosted in China at the Tencent Cloud Guangzhou Datacenter (AS45090) and uses an Intel Xeon Platinum 8255C CPU. The second is in the U.S. at AWS Oregon (AS16509) and uses an Intel Xeon E5-2686 v4 CPU. Reviewers can SSH into the servers using the credentials provided.

A.2.4 Software dependencies

Both servers provided to the reviewers run Ubuntu 20.04 LTS. The following software tools and libraries are required on the local machine to compile and run the tools and experiments in this artifact:

- GNU make utility
- GNU plot (gnuplot) utility
- Docker >= 20.10.0
- Python 3.9+

The primary evaluation method uses a Docker container to build all necessary tools. However, individual tools can be compiled and run directly on your local machine without Docker. If you choose this path, you will also need to install:

- Go 1.22+
- Rust (rustc,cargo) 1.82+
- Packages: `build-essential clang cmake`

A.2.5 Benchmarks

None.

A.3 Set-up

We provide AE reviewers with two pre-configured VPSes (usenix-ae-us and usenix-ae-cn) that include all the required tools and dependencies. For those who wish to set up their

own environment, we also offer a one-click script to automatically compile and transfer the tools on new VPSes.

A.3.1 Installation

- Docker: https://docs.docker.com/get-docker/
- GNU make: https://gnu.org/software/make/
- Download the repository: https://github.com/gfw-report/usenixsecurity25-quic-sni using Git clone.
- Compile and transfer tools to VPSes:

```
cd usenixsecurity25-quic-sni/utils;
make  \
  SERVER_HOST=$USENIX_AE_US \
  SERVER_USER=$USENIX_AE_US_USER \
  SERVER_SSH_KEY=$USENIX_AE_SSH_KEY \
  CLIENT_HOST=$USENIX_AE_CN \
  CLIENT_USER=$USENIX_AE_CN_USER \
  CLIENT_SSH_KEY=$USENIX_AE_SSH_KEY
```

Running the `make` command will compile the necessary tools and transfer tools to the servers. The `SERVER_HOST/USER/KEY` and `CLIENT_HOST/USER/KEY` variables should be set using the credentials provided.

A.3.2 Basic Test

Log in to the two VPSes using the provided credentials:

```
ssh -i $USENIX_AE_SSH_KEY \
  $USENIX_AE_CN_USER@$USENIX_AE_CN

# In a separate terminal window:
ssh -i $USENIX_AE_SSH_KEY \
  $USENIX_AE_US_USER@$USENIX_AE_US
```

Then run the following command on the usenix-ae-us VPS:

```
sudo tcpdump udp and host $USENIX_AE_CN \
 -Uw - | ./server-parser
```

On the usenix-ae-cn VPS:

```
echo baidu.com | ./quic-sni-sender  -p 443 \
  --sport=65000 --dip=$USENIX_AE_US \
  --socket-pool-size 1
```

After 10 seconds, check the terminal on the server (usenix-ae-us). You will see output in CSV format. The value in the last column indicates the test result. A value of `True` means the QUIC connection was blocked, while `False` means the connection was successful.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): GFW decrypts QUIC Initial packets to inspect SNI fields and block QUIC connections. This is proven by experiment (E1).

(C2): The GFW blocks QUIC connections where the client's source port is greater than the server's destination port. This is supported by experiment (E2).

(C3): The GFW exhibits varied responses to different QUIC and QUIC-like payloads. This can be confirmed by experiment (E3).

(C4): Sending a dummy UDP payload before the Client Initial packet can bypass the GFW's QUIC-SNI blocking. This is proven by experiment (E4).

A.4.2 Experiments

Before proceeding, ensure the VPSes are set up with the tools and dependencies detailed in the previous section. The experiments will be conducted between a client machine in China (usenix-ae-cn) and a server in the U.S. (usenix-ae-us). In each experiment, the client will send QUIC packets/probes to the server. To prevent residual interference, a 5-minute interval should be observed between each test. We recommend running the experiments multiple times, as the GFW's QUIC-blocking behavior can exhibit diurnal variations (see Section 3.4 of the paper).

(E1): *[Test QUIC-SNI Blocking] [5 human-minutes + 5 compute-mintues]: This experiment verifies the GFW's ability to decrypt and block QUIC connections based on SNI in the Initial Packet.*

Preparation: Log in to the two VPSes using the provided credentials.

Execution: On the usenix-ae-us VPS, run:

```
sudo tcpdump udp and host $USENIX_AE_CN \
-Uw - | ./server-parser
```

On the client (usenix-ae-cn), run the following command to send a QUIC probe. This probe consists of a Client Initial packet, followed by a 5 second delay, and then subsequent 10-byte UDP payloads.

```
echo google.com | ./quic-sni-sender -p 443 \
--socket-pool-size 1 --dip=$USENIX_AE_US \
--sport=55000
```

Results: Wait approximately 10 seconds for the server-parser script on the server to produce its output. A typical result will appear as follows:

```
# 2025/06/04 03:01:34 Started parsing: /dev/stdin
# tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
# 2025-06-04T03:02:08Z,{usenix-ae-cn_IP},{usenix-ae-us_IP},55001,4437,google.com,true
```

The parser determines that a connection is blocked if it receives the QUIC Initial Packet but none of the subsequent UDP payloads arrive. This outcome is indicated by the true value in the final column. A false value signifies that no blocking was detected. Test other SNI values like `baidu.com` (to verify behavior for exempt domains) and `cloudflare-dns.com`, `youtube.com` (to verify blocking for other domains).

(E2): *[Testing Rule: SourcePort > Destination Port] [5 human-minutes + 5 compute-minutes]: This experiment tests the GFW's filtering rule for QUIC Initial packets in which the source port must be greater than the desti-*

nation port.

Preparation: Run the parser `( sudo tcpdump udp and host $USENIX_AE_CN -Uw - | ./server-parser )` from E1 on the server (usenix-ae-us).

Execution: Run the following command on the client (usenix-ae-cn):

```
echo google.com | ./quic-sni-sender \
-p 5000 --socket-pool-size 1 \
--dip=$USENIX_AE_US \
--followup-payloads=10 \
--bind-ip=0.0.0.0 \
--sport=4000 \
--no-use-greater-srcports
# SrcPort (4000) < DestPort (5000)
-> Expected Result: NOT BLOCKED

echo google.com | ./quic-sni-sender \
-p 443 --socket-pool-size 1 \
--dip=$USENIX_AE_US \
--followup-payloads=10 \
--bind-ip=0.0.0.0 \
--sport=65000
# SrcPort (65000) > DestPort (443) \
-> Expected Result: BLOCKED
```

Results The results from the first command should show a `False` value in the last column, indicating that the connection was not blocked. The second command should show a `True` value, indicating that the connection was blocked due to the source port being greater than the destination port.

(E3): *[Test Different QUIC Payloads] [30 human-minutes + 30 compute-minutes]: This experiment evaluates the GFW's blocking behavior based on various QUIC payloads as detailed in Table 3 of the paper. The payloads can be generated using the `quic-packet-builder` utility in the `utils` directory.*

Preparation: Begin by capturing UDP traffic on the server (usenix-ae-us) using: `sudo tcpdump udp and host $USENIX_AE_CN`

Execution: On the client (usenix-ae-cn), run the following commands using the payloads files linked in the table here.

```
# Send QUIC payload using netcat
nc -u -q 0 -p 60001 $USENIX_AE_US 444 \
< ./payloads/exp1.bin
```

Then send follow-up payloads repeatedly to the same destination port using the same source port using the following command:

```
# Send arbitrary UDP payload to the server
echo "10101010101010101010" | xxd -r -p \
| nc -u -q 0 -p 60001 $USENIX_AE_US 444
```

Results: Depending on if any follow-up payload packets arrive at the server, one can determine if the connection was blocked or not. The results should match findings in Table 3 of the paper. Change the source and destination ports between each test to avoid residual interference.

(E4): *[Test Dummy Payload Bypass] [5 human-minutes + 5 compute-minutes]: This experiment shows how sending a dummy UDP payload before the Client Initial packet can bypass the GFW's blocking.*

Preparation: Capture UDP traffic on the server (usenix-ae-us) using: `sudo tcpdump udp and host $USENIX_AE_CN`

Execution: Run the following command on the client (usenix-ae-cn):

```
# Send an arbitrary UDP payload to the server
echo "00000000000000000000000000000000" \
| xxd -r -p | nc -u -q 0 \
-p 65535 $USENIX_AE_US 6126
```

Wait a few seconds and then send the QUIC packet containing a forbidden SNI (google.com in this example):

```
echo google.com | ./quic-sni-sender -p 6126 \
--socket-pool-size 1 --dip=$USENIX_AE_US \
--sport=65535
```

Results: Successful arrival of all subsequent packets on the 4-tuple `(src_ip, 65535, dst_ip, 6126)` at the server confirms the bypass.

A.5 Reproducing Paper Resources

To reproduce the figures and data presented in the paper, follow these instructions: First, set up your Python environment:

```
python3 -m venv venv
. venv/bin/activate
pip install -r requirements.txt
```

Each figure and table in the paper can be reproduced individually via the make command in its respective experiment directory. A comprehensive list of the commands required to reproduce each figure and table is provided in Tables 1 and 2. These commands should be run from the `experiments` directory.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

Section	Figure	Command	Output file
3.2 GFW's Blocking Latency	Fig 2 & 10	cd how-fast-gfw-blocks; make clean; make	how-fast-the-gfw-blocks.pdf, how-fast-the-gfw-blocks-boxplot.pdf
3.3 Blocking Rule: Source Port >= Destination Port	Fig 3 & 11	cd rule-srcport-greater-than-dst-port; make clean; make	heatmap-ports-401-450-step-1_heatmap.pdf, heatmap-ports-1-65000-step-1000_heatmap.pdf
3.4 Diurnal Blocking Pattern	Fig 4	cd diurnal-blocking; make clean; make	diurnal-timeseries-three-sources.pdf
3.6 Parsing Idiosyncrasies	Fig 5	cd what-triggers-blocking; make clean; make	quic_parse_heatmap.pdf
4. Monitoring the Blocklist over Time	Fig 6	cd sni-blocklist; make clean; make	domains-blocked-over-quic-weekly.pdf
4.1. Comparison with Other Blocklists	Fig 7	cd overlap-between-blocklists; make clean; make	venn-intersection-between-lists.pdf
5. GFW Degradation Attack	Fig 8	cd degradation-attack; cd Figure_8_experiments /AVG_exp23-22-20_sensitive-stressing_and_exp25-26-27_random-stressing; make clean; make	stressing_rates.eps

Table 1: This table lists the commands required to reproduce each figure from the main paper using the provided artifact. For each figure, the corresponding section, command, and output file are specified.

Table	Data sources	Command
1	./network-tap/data/tuple-count-2025-01-22-16-00.statistics-quic-conn.txt, ./network-tap/data/tuple-count-2025-01-22-16-00.statistics-udp-pkt.txt	—
2	./ttl-location/data/DNS/{city}-dns-and-traceroute-result.txt, ./ttl-location/data/QUIC/{city}-ttl_anon.pcap	—
3	./what-triggers-blocking/payloads, ./what-triggers-blocking/results.txt	—
4	./sni-blocklist/	`make clean && make`
5	./overlap-between-blocklists	`make clean && make`
6	./availability-attack/data/ec2/	`make clean && make`

Table 2: This table lists the data sources and commands required to reproduce each table from the main paper using the artifact repository.



USENIX Security '25 Artifact Appendix: <Ares: Comprehensive Path Hijacking Detection via Routing Tree>

Yinxiang Tao[1*], Chengwan Zhang[*], Changqing An[1],
Shuying Zhuang[2†], Jilong Wang[3,1], and Congcong Miao[4†]

[1]Institute of Network Sciences and Cyberspace, Tsinghua University, Beijing, China
[2]Zhongguancun Laboratory, Beijing, China
[3]Quan Cheng Laboratory, 250103, Jinan, Shandong, China
[4]Tencent
tyx23@mails.tsinghua.edu.cn, x10000zhang@gmail.com,
{acq,wjl}@cernet.edu.cn, zhuangsy@mail.zgclab.edu.cn, mccmiao@163.com

A Artifact Appendix

A.1 Abstract

In the Artifacts, we have included all the source code mentioned in the Open-Science section, specifically the complete source code of our proposed path hijacking detection approach Ares. All the data used in our experiments were obtained from publicly available datasets, and we have described how to access these data in Ares' README documentation. Additionally, we have shared the raw data necessary to reproduce the results presented in our paper.

Furthermore, we have included the source code of three other methods in the shared link, as we compared Ares' performance against these approaches in the paper. The README document outlines how we obtained the source code for these three methods. To ensure these methods can be fairly compared with Ares on our datasets, we made certain modifications and additions to their source code. Detailed usage instructions are provided in the README files within each method's respective folder in the shared repository.

A.2 Description & Requirements

Since our artifacts include not only the source code of Ares but also the code for three other methods (Metis, DFOH, and BEAM), we will separately specify the dependencies required for each of the four approaches.

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy and ethical concerns for evaluators while executing our artifact. All raw data used in experiments are publicly available. Our use of these data does not raise any security, privacy and ethical concerns. Additionally, the open-source code we have published is either obtainable from other public sources or has obtained consent from the original code creators.

A.2.2 How to access

The source code used in our experiments – including Ares and the methods compared with Ares – along with the raw experimental data, is publicly available at https://doi.org/10.5281/zenodo.15589806.

A.2.3 Hardware dependencies

Our experiment was conducted on a server with Intel(R) Xeon(R) Gold 5218R CPU@2.10GHz which has 80 cores.

To successfully run Ares, you will need a server with sufficient memory (better over 100GB, our experimental machine was equipped with 251GB RAM). Furthermore, to properly compare Ares with the other three approaches, you will need a GPU-equipped server with cuda for running models, as all three methods involve machine learning components. Moreover, you will need over 166GB disk space to store DFOH's database.

A.2.4 Software dependencies

Our experiment was conducted on a server running Ubuntu 20.04.1 LTS and Python 3.9.13.

Ares. To successfully run Ares, you need to install the dependencies listed in the README document, which include: Python dependencies specified in requirements.txt, The bgpdump toolkit, The BGPStream framework. The README provides direct links to installation instructions for these components. Additionally, to mitigate potential dependency conflicts (e.g., from pre-existing packages in the server environ-

ment prior to experimentation), we provide a requirements-full.txt file containing all explicitly tested package versions.
Metis. To compare Metis with Ares, you need to install Python dependencies listed in requirements file in its folder.
BEAM. To compare BEAM with Ares, you need to follow the instructions listed in its original readme document at 'BEAM/readme.md' to install dependencies.
DFOH. To compare DFOH with Ares, you will need to install Docker and Python dependencies. Instructions are included in DFOH's original readme document at `DFOH/dfoh_runner/README.md`.

A.2.5 Benchmarks

Ares. The data required by Ares in the experiments with this artifact are already documented in the 'Data' folder.
Metis. The data and model required by Metis in the experiments with this artifact are already documented in the 'Metis' folder and some of its sub-folders.
BEAM. The data and model required by Metis in the experiments with this artifact are already documented in the 'BEAM' folder and some of its sub-folders.
DFOH. The database required by DFOH in the experiments with this artifact can be obtained at https://dfoh.uclouvain.be/database.

A.3 Set-up

A.3.1 Installation

In the beginning, download all files at https://doi.org/10.5281/zenodo.15589806 and decompress all compressed files. You should receive a READ document along with five folders: Ares, Metis, DFOH, BEAM, and Data.
Ares. Instructions are included in readme document at `Ares/README.md`. To install dependencies, follow the instructions bellow:
1. Install bgpdump tool following instructions at https://github.com/RIPE-NCC/bgpdump/wiki. Note to add this tool to the system PATH.
2. Install libBGPStream following instructions at https://bgpstream.caida.org/docs/install/bgpstream.
3. Install Python dependencies using **`pip install -r requirements.txt`**.

This should lead you to run Ares. If there are any missing dependencies, you could supplement it or check `requirements-full.txt`.
Metis. Instructions are included in readme document at `Metis/README.md`. Install Python dependencies using **`pip install -r requirements`**.
BEAM. Original instructions are included in readme document at `BEAM/readme.md` and our instructions are at `BEAM/README.md`. Follow the original instructions to install dependencies. Note that if you want to use the model we provide and do not want to train one yourself, you do not need to create a virtual environment for BEAM.
DFOH. Original instructions are included in readme document at `DFOH/dfoh_runner/README.md` and our instructions are at `DFOH/README.md`. To install dependencies, follow the instructions bellow:
1. Install docker using installation guide at https://docs.docker.com/engine/install/ubuntu/.
2. Install Python dependencies using **`pip install -r requirements.txt`** at `DFOH/dfoh_runner/main`.
3. Download DFOH's database at https://dfoh.uclouvain.be/database and store it in `DFOH/DFOH_db`. For DFOH, the original data acquisition method described in its documentation may no longer be functional. As an alternative, you may need to implement a web crawler to retrieve this database. While we provide our original crawler script (`download_db.py`), please note that the script requires modifications to properly download subdirectory contents from the target database.
4. Pull docker images or build it yourself. The DFOH implementation includes functionality for pulling Docker images automatically, but to circumvent potential deployment obstacles, we recommend manually building the Docker images using DFOH's provided scripts. Specifically, you should run **`bash build_and_upload.sh`** at each directory except `dfoh_runner` (Bypass the login step so the script only builds the images without attempting to upload them).

A.3.2 Basic Test

After completing the installation, perform basic testing for each method using the following procedures.
Ares. Run **`python3 main_simulation.py 4 2`** and **`python3 main_his.py 7 03 97 50`** at `Ares` folder to check whether Ares can successfully function. If output files are generated in `Ares/running_log` and `Ares/simresults`, it means Ares is functioning fine.
Metis. Run **`python3 predict.py 4 2`** at `Metis/simulation` to check whether Metis can successfully function. If output are generated in files in `Metis/simulation/results`, it means Metis is functioning fine.
BEAM. Run **`python3 run_on_dataset.py 4 2`** at 'BEAM' folder to check whether BEAM can successfully function. If output are generated in files in `BEAM/simulation` and `BEAM/simulation_results`, it means BEAM is functioning fine.
DFOH. Run **`python3 run_on_dataset.py 4 2`** at `DFOH/dfoh_runner/main` to check whether DFOH can successfully function. If output are generated in files in `DFOH/simulation` and `DFOH/simulation_results`, it means DFOH is functioning fine.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Ares is effective against reported historical hijacking events and produces a few number of alarms. This is proven by E1 described in Section 5.1 whose results are illustrated/reported in Table 4.

(C2): Ares has better recall rate on our simulated hijacking event dataset than existing state-of-the-art methods. This is proven by E2 described in Section 5.2 whose results are illustrated/reported in Table 5-6, 11-16.

(C3): Ares comparable false positive rate with existing state-of-the-art methods on our sampled normal route change dataset. This is proven by E3 described in Section 5.2 whose results are illustrated/reported in Table 7.

(C4): Ares has lower runtime overhead than existing state-of-the-art methods. This is proven by E4 described in Section 5.3 whose results are illustrated/reported in Table 8.

A.4.2 Experiments

(E1): [20 human-minutes + 3+ compute-hour + 60GB memory]: Apply Ares to 12 collected historical events and find that it can successfully detect all events with a few number of alarms.
Preparation: Correctly obtain and position the Data folder contents from the provided artifact.
Execution: Run **`python3 run_his_script.py <start_index> <end_index>`** at `Ares` folder to detect each historical event using our used settings one by one.
Results: Results are stored at `Ares/running_log`. Run **`python3 check_alerts.py`** to obtain results presented in Table 4. The outcomes are expected to closely approximate the results reported in the paper.

(E2): [1 human-hour + 20-30 compute-hour if parallel executed (hundreds of compute-hour if not)]: Apply Ares as well as Metis, BEAM and DFOH to our simulated hijacking events to compare their recall rate.
Preparation: Correctly obtain and position the Data folder contents from the provided artifact. Ensure the dependencies for each method are installed and all of them can function.
Execution: Run all commands in `run_simulationdetect.sh` at `Ares`, `run_on_dataset.sh` at `BEAM` and `DFOH/dfoh_runner/main`, `run_predict.sh` at `Metis/simulation` respectively to perform the detection on simulated events (If your CPU has sufficient multi-core capabilities, you may attempt parallel execution of these commands to significantly reduce processing time).
Results: Run **`python3 check_dataset.py`** for each methods to get recall rate results. Note that due to involving machine learning, the results of Metis and DFOH may have slight differences when executed repeatedly. The outcomes are expected to show that Ares has better recall rate than other methods.

(E3): [20 human-minute + 1-2 compute-hour]: Apply Ares as well as Metis, BEAM and DFOH to sampled normal route changes to compare their false positive rate.
Preparation: Correctly obtain and position the Data folder contents from the provided artifact. Ensure the dependencies for each method are installed and all of them can function.
Execution: Run **`python3 main_normal.py > running_log/detect_normal.txt`** at `Ares`, **`python3 run_on_normal.py`** at `BEAM`, `DFOH/dfoh_runner/main`, and `Metis/simulation` respectively to perform the detection on normal changes.
Results: Run **`python3 check_normal.py`** for Ares and manually inspect results for other three methods to compare their false positive rate. Note that BEAM had two results for two different thresholds (details see Section 5.2). The outcomes are expected to show that all methods have comparable false positive rate.

(E4): [20 human-minute + 5-6 compute-hour]: Apply Ares as well as Metis, BEAM and DFOH to a same amount of BGP updates to compare their efficiency.
Preparation: Correctly obtain and position the Data folder contents from the provided artifact. Ensure the dependencies for each method are installed and all of them can function. Run **`getUpdate.py`** at `Data/updateData` to obtain BGP updates.
Execution: Run **`python3 eff_test.py`** at `Ares`, **`python3 routing_monitor/detect_route_change_riperis.py --collector rrc03 --year 2024 --month 10`** at `BEAM`, **`python3 my_run_daily.py --date 2024-10-01 --date_end 2024-10-02 --db_dir <absolute path of DFOH_db>`** at `DFOH/dfoh_runner/main`, **`python3 eff.py`** at `Metis/simulation` respectively to perform the comparison.
Results: Run **`python3 eff_check.py`** for Ares, Metis, BEAM and manually inspect results for DFOH to compare their runtime overhead (DFOH does not detect update by update, therefore only can be counted in total). The outcomes could vary due to running devices, but Ares' runtime overhead is expected to be lower than the other methods.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Trust but Verify: An Assessment of Vulnerability Tagging Services

Szu-Chun Huang, Harm Griffioen, Max van der Horst,
Georgios Smaragdakis, Michel van Eeten, and Yury Zhauniarovich

Delft University of Technology

A Artifact Appendix

A.1 Abstract

The artifact consists of two main parts: 1. **Modified Nuclei Templates**: These modified templates were used in the study as described in the Methodology section of the paper. 2. **Vulnerable Docker Environments**: This part includes several vulnerable and non-vulnerable Docker environment files for the Ground Truth-Based Evaluation section.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact does not present any security, privacy, or ethical concerns for evaluators.

A.2.2 How to access

The artifact is permanently available at https://doi.org/10.5281/zenodo.14732150.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

None.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Users can install Nuclei from ProjectDiscovery's GitHub repository[1]. One can download the latest binary from *Nuclei Releases*[2]. Next, users can build the Docker images by running the *docker_build.sh* script with command: `sh docker_build.sh` and create a Docker network named *bridge-honeypot* with command: `docker network create bridge-honeypot`.

A.3.2 Basic Test

To test the functionality of Nuclei templates, users can choose a target endpoint to scan or use the vulnerable Docker environments provided in the *Vulhub repository*[3] to build a vulnerable endpoint. Also, users can run vulnerable endpoints by using the Docker Compose commands at the top of each *docker-compose* file in the **Vulnerable Docker Environments** folder.

Then, users can run the Nuclei templates against the target endpoint using the following command: `[Path to Nuclei binary] -u [Target endpoint] -t [Path to Nuclei template]`.

A.4 Evaluation workflow

None.

A.4.1 Major Claims

None.

A.4.2 Experiments

None.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

[1] https://github.com/projectdiscovery/nuclei

[2] https://github.com/projectdiscovery/nuclei/releases

[3] https://github.com/vulhub/vulhub



USENIX Security '25 Artifact Appendix: Catch-22: Uncovering Compromised Hosts using SSH Public Keys

Cristian Munteanu
Max Planck Institute for Informatics

Georgios Smaragdakis
Delft University of Technology

Anja Feldmann
Max Planck Institute for Informatics

Tobias Fiebig
Max Planck Institute for Informatics

A Artifact Appendix

A.1 Abstract

This paper presents an Internet-scale scanning tool designed to identify compromised SSH servers by analyzing SSH behavior during public key authentication. Our tool detects malicious SSH keys by sending authentication challenges without accessing the compromised systems or requiring audit-level privileges. This approach ensures a non-intrusive and ethical method for assessing exploited SSH servers.

A.2 Description & Requirements

This section describes our scanning tool, including hardware and system requirements, as well as ethical considerations. Additionally, we provide installation and testing instructions.

A.2.1 Security, Privacy, and Ethical Considerations

Our study uses a set of public keys known to be involved in malicious activities. These keys were shared with us through our collaboration with Bitdefender and cannot be publicly disclosed. Revealing these keys would diminish the effectiveness of our scans, as malicious actors could identify and replace their compromised keys. However, some of these keys are publicly available and can be found through a simple web search. For Experiment 1 A.4.2, we have additional resources, including IP prefixes, IP addresses, and SSH keys, which cannot be publicly disclosed. These materials will be provided to the evaluation committee during the evaluation process.

On another note, the nature of the artifact must be carefully considered. The scanning tool, particularly when used at a large scale, can place significant pressure on the scanned networks. This may lead to unintended consequences, including unforeseen disruptions and dissatisfied system administrators. To mitigate these risks, it is essential to maintain a blocklist, ensure 24/7 contact availability, and carefully regulate the scan rate. Additionally, for an Internet-wide scan, obtaining approval from an Institutional Review Board (IRB) is mandatory.

A.2.2 Accessing the Artifact

The artifact is available at: https://edmond.mpg.de/dataset.xhtml?persistentId=doi%3A10.17617%2F3.LVPCS6

A.2.3 Hardware Requirements

The recommended *minimum* system specifications:
- 4 CPU cores
- 4 GB of RAM
- An active Internet connection

For a *full-scale Internet-wide scan*, we recommend using at least four machines, each with the following specifications:
- 16 core AMD EPYC 7232 or better
- 64 GB memory
- 10gbit network connection
- at least 4TB NVMe in raid 1

A.2.4 Software Requirements

Our scanning tool was tested and deployed on Debian 12. It requires ZMap [1] for port scanning and a patched version of ZGrab2 (included in the artifact). The scanning script will automatically download ZMap and build the patched ZGrab2.

The following software dependencies are required:

- Go (version 1.19 or later) – Install using: `sudo apt-get install golang` (Debian 12);
- Bash (`/bin/bash`) – Installed by default on a standard Debian 12 system.

A.2.5 Benchmarks

None.

[1] https://github.com/tumi8/zmap

A.3 Set-Up

Follow these steps to set up the scanning tool:

- Download the *scanner.tar.gz*
- Extract the tarball using: `tar -xvf scanner.tar.gz`.
- Navigate to the "scanner" directory. This directory contains the `scan_ssh_ip.sh` script and the `Zgrab2_patched` folder, which holds the patched ZGrab2 source code.

The contents of the "scanner" directory should look like this:

```
$ ls -alh scanner
drwxr-xr-x .
drwxr-xr-x ..
drwxr-xr-x Zgrab2_patched
-rwxr-xr-x scan_ssh_ip.sh
```

A.3.1 Installation

The primary dependency is Go. To install Go on Debian 12, run:

```
sudo apt-get install golang
```

A.3.2 Basic Functionality Test

Ensure you are in the "scanner" directory.

Execute:

```
./scan_ssh_ip.sh -help
```

Expected output:

```
Go is installed.
go version go<version>
Usage: ./scan_ssh_ip.sh [OPTIONS]

Options:
...
```

A.4 Evaluation Workflow

This section outlines the necessary steps to evaluate the functionality and reproducibility of our artifact.

A.4.1 Major Claims

The primary contributions of our tool are:

(C1): The scanning tool offers a reliable solution for determining whether a specific SSH public key is installed for a particular user on a publicly accessible SSH server.

(C2): Using this tool in a global scan, we detected over 21,700 unique compromised systems spanning 1,649 ASes and 144 countries, affected by 52 verified malicious keys provided by a threat intelligence partner. The scan uncovered critical Internet infrastructure vulnerabilities, highlighted campaigns such as the 'fritzfrog' IoT botnet and actors like 'teamtnt', and identified state-associated malicious keys within sensitive networks. By correlating findings with honeypot data, we revealed disparities in attacker activity representation, including underestimated APT operations.

A.4.2 Experiments

Experiment 1: Controlled environment This experiment will validate our first claim (C1) regarding the reliability of the SSH public key scanning solution. For the experiment, we deployed multiple SSH servers preconfigured with specific SSH keys. These test servers will be accessible only during the artifact evaluation. Additional details, including IP prefixes, SSH public/private keys, and configuration files, will be provided to the evaluation committee (see Section A.2.1).

Each test system is configured with various user accounts ('root', 'test', 'foo', and 'bar') and a predefined set of SSH keys. To facilitate testing, we provide:

- A list of IPv4 prefixes and IPv6 addresses: `ipv4_prefix_list` and `ipv6_list`;
- A collection of SSH public keys for testing: `mal_k_round_0`, `mal_k_round_1`, `mal_k_round_2`;
- A "ground-truth" file to compare experimental results: `sample_output.csv`;
- A shell script to streamline the experiment (Figure 1): `demo_run.sh`;
- A set of private SSH keys for non-root users to validate the scanner's capability. If the scanner detects a public key for a user on an SSH server, the corresponding private key can be used to establish a connection. Note that 'root' user keys are not shared due to security concerns: `demo_keys/`.

To conduct the experiment, you will need a machine–either a physical system or a virtual machine (VM)–that meets the recommended *minimum* system specifications (see Section A.2.3) and runs Debian 12. Follow these steps to run the experiment:

(Step 1) Place all additional resources, provided for the evaluation, in the same directory as the scanner script ("scanner").

(Step 2) Ensure the script "demo.sh" is executable: `chmod +x demo.sh`

(Step 3) Run the script: `./demo.sh`

After adding the additional resources, the contents of the "scanner" directory should look like this:

```
$ ls -alh scanner
drwxr-xr-x .
drwxr-xr-x ..
drwxr-xr-x demo_keys
drwxr-xr-x Zgrab2_patched
-rwxr-xr-x demo_run.sh
-rw-r--r-- ipv4_prefix_list
-rw-r--r-- ipv6_list
-rw-r--r-- mal_k_round_0
-rw-r--r-- mal_k_round_1
-rw-r--r-- mal_k_round_2
-rwxr-xr-x scan_ssh_ip.sh
```

After the script execution, the results can be found the `demo_run.csv` file. The Zmap and Zgrab2 output files can be found in the `ssh_scan_results_v4/` and `ssh_scan_results_v6/` folders.

Experiment 2: Internet-Compromised SSH This experiment will validate our second claim (C2) regarding the feasibility of conducting a full-scale Internet scan and accurately identifying SSH servers compromised by known malicious SSH keys. The data for our study was generated through this experiment.

For a *full-scale Internet-wide scan*, we recommend using at least four machines, each meeting the specifications outlined in Section A.2.3. We strongly advise using physical machines rather than virtual machines (VMs) to prevent potential network-related issues. A VM may struggle to handle a sustained high volume of packets, potentially biasing the results.

To replicate this experiment, we define a series of steps. However, some of these steps are not strictly deterministic and cannot be precisely reproduced; they should be considered as guidelines. Nevertheless, each step is crucial and must be executed.

It is important to note that certain steps may take an unpredictable amount of time, affecting the overall duration of the experiment. For example, the Institutional Review Board (IRB) approval process can take anywhere from two weeks to six months, depending on the IRB's policies and workload. Additionally, the IRB may impose limitations on the scanning speed (e.g., restricting scans to no more than 50,000 packets per second).

As a benchmark, in our experiment, obtaining ethical approval took over three weeks. Conducting a full IPv4 scan for a single user with 52 keys required approximately 13 days, followed by an additional 3-4 days to scan the IPv6 addresses in the `ipv6hitlist`.

Follow these steps to run the experiment:

(Step 1) Obtain SSH public keys. Since the public keys we used cannot be shared (see Section A.2.1), you can instead source malicious SSH public keys from publicly available locations such as security blogs or public databases of Indicators of Compromise (IOCs). For instance, a widely known malicious SSH public key can be identified by its label–'mdrfckr'–which is referenced in our paper as `MK06`.

(Step 2) Define the target IP address range. For IPv4 scanning, you can use the '0.0.0.0/0' prefix to cover the entire IPv4 address space. For IPv6, consider using the `ipv6hitlist` service to obtain a list of active IPv6 addresses.

(Step 3) Build an IP blocklist for IPv4 and IPv6.

- For IPv4, a good starting point is the IANA IPv4 Special-Purpose Address Registry, which contains non-routable and reserved addresses.
- Store the blocklist prefixes in a separate file and specify it when running the scan script.
- For IPv6, since entire prefixes cannot be scanned, filtering must be done at the individual IP level. Begin by removing IPs listed in the IANA IPv6 Special-Purpose Address Registry. Tools such as `grepcidr` can assist in this process.

(Step 4) Obtain Institutional Review Board (IRB) approval. For large-scale scans, obtaining IRB approval is mandatory before execution. The IRB committee will assess the ethical implications of the experiment and determine the appropriate scanning rate while ensuring compliance with best practices (Menlo Report [2], Drumenic et al.). Setting up informative rDNS, maintaining accurate WHOIS records, hosting a website that explains your scanning activities, and ensuring 24/7 availability via phone and email would constitute the requirements for adhering to best practices.

(Step 5) Run the scan. Start the scan using the script provided in the artifact. Throughout the scanning process, ensure that the blocklist remains up to date to prevent unintended scanning of restricted or sensitive IP addresses.

A.5 Notes on Reusability

This artifact is designed for long-term reusability. It provides the necessary tools to detect SSH-compromised hosts across the Internet. Anyone collecting malicious SSH public keys can use this artifact to identify systems compromised by those keys.

A comprehensive 'README' file is included in the artifact, detailing:

[2] https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_1.pdf

```
#!/bin/bash

OUT_FILE="demo_run.csv"

# Check if jq is installed
if command -v jq &> /dev/null; then
    echo "JQ is installed."
else
    echo "Installing JQ..."
    sudo apt-get update && sudo apt-get -y install jq
fi
jq --version

echo "====== Starting Demo ======"
echo "Estimated run time per user: ~3m40s"
echo "Total estimated time: ~30 min"

# Run scans for different users over IPv4
for user in foo bar test root; do
    ./scan_ssh_ip.sh -ipwhitelist=ipv4_prefix_list -port=22 -user=$user -scantype=4
       -wait=10
done

# Run scans for different users over IPv6
for user in foo bar test root; do
    ./scan_ssh_ip.sh -ipset=ipv6_list -port=22 -user=$user -scantype=6 -wait=10
done

echo "IP,USER,Key" > $OUT_FILE
cat ssh_scan_results*/*/key_* | grep "key-accepted\":true" | jq -r '.ip + ",
   " + .data.ssh.result.dummyauth.user + ",
   " + .data.ssh.result.dummyauth.key' >>
   $OUT_FILE
echo "Demo run finished. Output file: $OUT_FILE"
```

Figure 1: SSH script for demo run.

- How to execute a scan
- How to adjust scan parameters (e.g., wait times, network interface selection)

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Exposing the Guardrails: Reverse-Engineering and Jailbreaking Safety Filters in DALL·E Text-to-Image Pipelines

Corban Villa
New York University Abu Dhabi

Shujaat Mirza
New York University

Christina Pöpper
New York University Abu Dhabi

A Artifact Appendix

A.1 Abstract

This artifact includes all of the required code and dataset to validate and reproduce results of the paper: *Exposing the Guardrails: Reverse-Engineering and Jailbreaking Safety Filters in DALL·E Text-to-Image Pipelines*. This full dataset, which thoroughly evaluates DALL·E 2/3 systems across thousands of requests, is fully available to elicit queries and analyze (with harmful prompts redacted but available upon request). We additionally include scripts to perform a scaled-down experiment to allow evaluators to independently assess the reproducibility of our results.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact has been carefully constructed to prevent unintended exposure to potentially harmful or disturbing material inherent to the nature of this work. For instance, we redact image prompts in our public dataset–replacing them with SHA256 hashes. (Unredacted prompts are available upon request.) Artifact evaluation scripts are entirely automated and do not require evaluators to interact with any potentially harmful material. Images produced are stored directly in the database to prevent inadvertent exposure.

We do not cause harm to the T2I systems as we are using them as a regular user within the existing rate limits. Furthermore, our methodology is consistent with previous peer-reviewed work that evaluates T2I systems under adversarial conditions [1, 2].

A.2.2 How to access

Artifact: The dataset and code is permanently available at: https://doi.org/10.5281/zenodo.14735417.

GitHub: The code and datasets above are additionally available on GitHub.

A.2.3 Hardware dependencies

A commodity computer:

- 2+ CPU cores
- 4GB+ memory
- 10GB+ disk storage

A.2.4 Software dependencies

Operating System Requirements: Evaluation scripts are validated with `Ubuntu 22.04`. Compatibility with other operating systems may vary.

Software Package Requirements: `docker` and `docker compose` (install guide). Validated with version `27.3.1`.

Privileged Access Requirements: A few of our configuration scripts require the user have sudo access in order to manage container data. For instance, the `postgres` data is owned by root within the container, and thus requires sudo privileges to remove the data after use. If this is a concern, we recommend utilizing a virtual machine for experiments.

A.2.5 Benchmarks

Adversarial prompt datasets originate from Qu et al., Rando et al., and Yang et al. These prompts do not need to be downloaded by evaluators.

A.3 Set-up

A.3.1 Installation

Install the latest version of docker and add your current user to the `docker` group:

- `curl -L https://get.docker.io | bash`
- `sudo usermod -aG docker $USER`
- Logout and log back in.

Clone and enter the GitHub repository:

- `git clone https://github.com/corbanvilla/T2I-Attacks-USENIX-2025.git`
- `cd T2I-Attacks-USENIX-2025`

A.3.2 Basic Test

Execute: `./scripts/validate-install.sh`

```
Command:
> ./scripts/validate-install.sh

Expected Output:
...
Docker compose build completed successfully
All validation checks passed!
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We reverse-engineer the black-box cascading safety guardrails in DALL·E models using a novel time-based side-channel, providing insights into a multi-stage filtering process (§5). This is replicated in experiment (E1), described in §5.1, which identifies the presence of multiple DALL·E 2 filters through statistical significance testing. Experiment results also mirror those in Fig. 2.

(C2): We synthesize key takeaways for T2I system security by juxtaposing safety mechanisms present in DALL·E 2 and DALL·E 3, notably the incorporation of an LLM-based implicit filter in DALL·E 3 (§4.4) to soften harmful prompts, quantified through experiments that analyze prompt toxicity before and after revisions (E2).

(C3): We introduce novel jailbreaking attacks specific to T2I models, namely T2I negation and low-resource-language attacks, which exploit the limitations of safety filters in handling negated phrases and less common languages (§6). We experimentally evaluate the success rates of jailbreaking low-resource languages (E3).

A.4.2 Experiments

Experiment Data Collection: [30-45 compute-minutes]: To run all experiments (E1-E3) and collect the required dataset for analysis, run the following commands. Experiments are estimated to require 30-45 minutes to complete. Please allow the experiments to *run continuously and uninterrupted to completion* to minimize the potential for noise variation in the DALL·E system performance.

The script will prompt evaluators to provide three key configuration secrets, which are included via a HotCRP comment: **1)** An OpenAI API key, **2)** an OpenAI organization key, and **3)** a prompt dataset decryption key.

Execute: `./scripts/artifact-evaluation.sh`

Results: Experiments are validated using Jupyter Notebook analysis scripts. Connect to the pre-configured Jupyter notebook server at: http://127.0.0.1:8888/lab (The default security token is: `T2I-USENIX-rubpGTNAzgd4tgk7npb`)

(E1): [DALL·E 2 Blocklist Probing] [5 human-minutes]:

```
Command:
> ./scripts/artifact-evaluation.sh

Expected Output:
...
All experiments completed
Artifact evaluation completed
```

Description: This experiment demonstrates the effectiveness of our DALL·E 2 blocklist probing attack, which allows an attacker to detect independent filter systems by their respective response times (§5.1).
Dataset: This experiment uses 5 pairs of blocklisted and mutated words, repeated 5 times each, for a total of 50 requests to DALL·E 2.
Success Criteria: The experiment demonstrates the statistical significance between rejection timings.
Results: To validate our blocklist probing side-channel attacks:

1. Open `src/artifact_evaluation/1-timing-side-channel.ipynb` in the Jupyter notebook.
2. Select `Run All Cells` from the `Run` menu.
3. Validate the final cell demonstrates statistical significance.

(E2): *[Quantifying Prompt Softening] [5 human-minutes]:*
Description: This experiment quantifies the effect of the LLM prompt revision prompt as an implicit filtering mechanism in DALL·E 3 using two key metrics: Toxicity Absolute Change and Toxicity Theme Similarity (§3.3).
Dataset: This experiment uses 5 prompts in 4 languages (Hawaiian, Lao, Nepali, Sinhala) for a total of 20 requests to DALL·E 3. Prompts and revised prompts are evaluated for toxicity using the OpenAI Moderation API.
Success Criteria: The experiment demonstrates that the prompt revision model "softens" the toxicity of harmful prompts to various degrees of success under multilingual inputs.
Results: To validate our prompt softening claims:

1. Open `src/artifact_evaluation/2-prompt-softening.ipynb` in the Jupyter notebook.
2. Select `Run All Cells` from the `Run` menu.
3. Validate the final cell demonstrates various toxicity changes and similarity decreases, with the magnitude of change dependent on the respective language.

(E3): [Low-Resource Languages] [5 human-minutes]:
Description: This experiment demonstrates that low-resource languages can be used as an effective mechanism to bypass DALL·E safety filters (§6.1).
Dataset: This experiment uses 5 prompts in 5 languages (English, Nepali, Sinhala, Lao, Hawaiian) for a total of 25 requests to DALL·E 3.

Success Criteria: Languages with fewer pages in the Common Crawl dataset frequently feature higher prompt acceptance rates.

Results: To validate our low-resource language attacks:

1. Open `src/artifact_evaluation/3-low-resource-lang-attack.ipynb` in the Jupyter notebook.
2. Select `Run All Cells` from the `Run` menu.
3. Validate the final cell depicts various language bypass rates.

A.4.3 Cleanup Procedure

To shutdown and remove the deployed containers and their respective images, run the following script. After executing, you may additionally wish to remove (`rm -rf`) the repository. Execute: `./scripts/cleanup.sh`

A.5 Notes on Reusability

We encourage future research to leverage our comprehensive datasets. The full experiment dataset is published under `datasets/postgres/redacted.sql.zip`. The `README.md` documentation describes how to interact with the dataset more comprehensively. Moreover, `src/examples.ipynb` details a number of ways to query the database, including how to use specific functions such as cosine similarity or vector norms.

While specifics of the dataset, such as average or median response times may vary as DALL·E 2/3 are continuously updated by OpenAI, we expect the core findings to maintain relevancy (e.g., the statistical distributions of response times). **Exploring Key Findings:** All tables and figures in this work can be produced using notebook files located under: `src/paper`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Self-interpreting Adversarial Images

Tingwei Zhang[†] Collin Zhang[†] John X. Morris[†] Eugene Bagdasarian[§] Vitaly Shmatikov[†]
[†]Cornell Tech [§]University of Massachusetts Amherst
{tingwei, collinzhang, jxm3}@cs.cornell.edu eugene@cs.umass.edu shmat@cs.cornell.edu

A Artifact Appendix

A.1 Abstract

This artifact provides the implementation and evaluation framework for the paper *Self-Interpreting Adversarial Images*, which introduces a novel class of indirect, cross-modal prompt injection attacks on Visual Language Models (VLMs). These attacks embed hidden meta-instructions in natural-looking images to covertly influence model outputs toward adversary-specified meat-objectives (e.g., sentiment, political bias, language, or style), while maintaining plausible and coherent responses.

The artifact includes code, data, and instructions to reproduce the paper's core results on MiniGPT-4, LLaVA, and InstructBLIP. It supports generation of adversarial image soft prompts, evaluation under diverse meta-objectives, preservation of semantics, and robustness testing across perturbation norms and defense strategies.

This artifact ensures reproducibility, facilitates research on cross-modal attacks, and provides a practical framework for evaluating VLM vulnerabilities and defenses.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The goal of this work is to support the safe deployment of VLMs and to encourage research on defenses against cross-modal prompt injection attacks. All datasets and models used in this artifact are publicly available, and the artifact itself is non-destructive and intended solely for research and evaluation purposes.

A.2.2 How to access

The code is available at this URL: https://figshare.com/articles/software/Self-interpreting_Adversarial_Images/29219051?file=56929889

A.2.3 Hardware dependencies

The artifact evaluation requires the following hardware configuration:

- **CPU**: We used dual Intel(R) Xeon(R) Gold 6448Y processors with 64 cores in total, but comparable hardware will likely suffice.
- **Memory**: Our machine had 256 GiB of system RAM, however much less will suffice.
- **GPU**: Our experiments were run on either two NVIDIA A40 GPUs (each with 48 GB of memory) or a single NVIDIA A100 GPU with 80 GB of memory. Smaller GPUs may still work, but will likely require a reduced batch size.
- **Storage**: At least 40 GiB to store datasets, model checkpoints, and experiment results.

A.2.4 Software dependencies

The artifact requires the following operating system and essential software packages:

- **Operating System**: The code has been tested on Ubuntu 20.04 LTS. It is recommended to use this OS for compatibility and to ensure reproducibility of the results. Other Linux distributions may work but are not guaranteed.
- **Python**: Python 3.9 or higher is required. It is recommended to use a virtual environment (such as Conda) to manage dependencies. For better compatibility and to avoid package conflicts, install the dependencies for each model independently.
- **CUDA and cuDNN**: To utilize GPU acceleration, ensure that CUDA 11.0 or higher and cuDNN are installed.

A.3 Set-up

A.3.1 Installation

Follow the **Setup** instructions in the `README.md` file of our GitHub repository to download the model checkpoints for

each model. Additionally, set up the environment according to the instructions provided in each model's repository.

A.3.2 Basic test

Activate your conda environment, then run the following commands:

1. Generate an image soft prompt:

```
python minigpt_visual_attack.py \
    --data_path instruction_data/coco_1/
    Sentiment/dataset.csv \
    --instruction negative \
    --image_file clean_images/coco_1.jpg \
    --save_dir output/example
```

2. Perform inference on the generated image:

```
python -u minigpt_inference.py \
    --gpu_id 0 \
    --data_path instruction_data/coco_1/
    Sentiment/dataset.csv \
    --image_file output/example/bad_prompt.bmp \
    --output_file output/example/result.jsonl
```

You can customize the attack by replacing the image, dataset, model, and instruction parameters with your own choices. The model's response will be saved as `result.jsonl` in the specified output directory.

A.4 Evaluation Workflow

A.4.1 Major claims

(C1): We evaluate our method on the available open-source VLMs with meta-instructions corresponding to different meta-objectives and show that **image perturbations encoding meta instructions are as effective as steering models' outputs as explicit instructions**. This corresponds to Table 2 in the paper. This is proven by experiment (E1).

(C2): We demonstrate that **meta-instructions preserve image semantics**. We use several metrics, including embedding and structural similarity(SSIM) and oracle LLM evaluation, to show that target VLMs' responses are based on the visual content of input images. This corresponds to Table 3 and Table 4 in the paper. This is proven by experiments (E2) and (E3).

(C3): We analyze stealthiness of our method and performance under different perturbation norms, demonstrating that **our method remains effective across various perturbation constraints**. This corresponds to Table 5 in the paper. This is proven by experiment (E4).

(C4): We analyze **transferability of image soft prompts**, showing that the attack can be effective even when the adversary does not know which specific VLM the victim will be applying to the adversary's images. This corresponds to Table 6 in the paper. This is proven by experiment (E5).

(C5): We survey several **countermeasures** against our attack based on feature distillation and anomaly detection based on the embedding consistency of augmentations. This corresponds to Table 7 and Table 8 in the paper. This is proven by experiments (E6) and (E7).

A.4.2 Experiments

(E1): *[Image soft prompt generation and evaluation] [2 human-hours + 200 compute-hours (can be reduced with more or larger GPUs, or run part of the experiments)]: This experiment generates the results of Table 2 in the paper.*
Execution: Read `script/README.md` for generating and evaluating single images. To reproduce the entire Table 2, run all shell files in `script/` with the naming format `script_{instruction}_{model}`, which include the code for generating attacks, inferencing models with image soft prompts, and baselines.
Results: The model output will be saved to JSON files. Run the Jupyter notebook `eval_instruction_following.ipynb` with the correct paths to evaluate the attack success.

(E2): *[Image semantics preservation - embedding and SSIM similarity] [1 human-hour + 2 compute-hours]: This experiment generates the results of Table 3 in the paper.*
Execution: For embedding and SSIM similarity analysis, use the `eval_content_preserving.ipynb` notebook. Update the "model_name" and "sim_metric" parameters in the first code block, and ensure all file paths are correctly configured.
Results: The similarity metrics will be computed and presented in the notebook. Embedding similarity results fill columns 1, 3, and 5 of Table 3, while SSIM similarity results fill columns 2, 4, and 6 of Table 3.

(E3): *[Image semantics preservation - oracle LLM evaluation] [2 human-hours]: This experiment generates the results of Table 4 in the paper.*
Execution: For columns 1, 4, and 7: Query the target VLM whether the label accurately represents the content of the perturbed image. The script can be found in the same script as (E1) in the inference stage, with the commented header "Content preservation evaluation". The result will be saved in the same directory as the image soft prompt. Run the `eval_content_preserving.ipynb` notebook; the second block will generate the result.
For columns 2, 5, and 8: These depict the metric of output relevant to clean image. For columns 3, 6, and 9: These depict the metric of output relevant to perturbed image. We take the clean image/perturbed image and the target model's response to clean/perturbed image and use the

following query: "With yes or no, determine if the output is relevant to the image and answers the question?". Look for the percentage of yes in the result.
Results: For columns 1, 4, and 7: The results will be displayed in the notebook. For the result columns: The result needs to be manually calculated.

(E4): *[Stealthiness analysis under different perturbation norms] [1 human-hour + 600 compute-hours (can be reduced with more or larger GPUs, or run part of the experiments)]: This experiment generates the results of Table 5 in the paper.*
Execution: Navigate to `script/minigpt4/minigpt4_coco_script/` and execute the constraint-specific scripts: `script_sentiment_minigpt4_16.sh` (L_∞ norm, $\varepsilon = 16$), `script_sentiment_minigpt4_l2_12.sh` (L_2 norm, $\varepsilon = 12$), `script_sentiment_minigpt4_l2_24.sh` (L_2 norm, $\varepsilon = 24$), and `script_sentiment_minigpt4_l2_6.sh` (L_2 norm, $\varepsilon = 6$).
Results: Follow the same evaluation method as (E1) using the evaluation notebook, updating the file paths to point to the corresponding constraint-specific result directories. The notebook evaluation will display attack success rates across different perturbation constraints.

(E5): *[Transferability analysis] [1 human-hour + 2 compute-hours]: This experiment generates the results of Table 6 in the paper.*
Execution: Execute `script/minigpt4/minigpt4_coco_script/script_sentiment_minigpt4_transfer.sh` to perform inference on different models using image soft prompts generated by MiniGPT-4. Query GPT-4o with the same image soft prompts to obtain additional results.
Results: Evaluate the results using `eval_instruction_following.ipynb` with correct file paths. For GPT-4o results, perform manual calculation.

(E6): *[Feature Distillation defense] [1 human-hour + 2 compute-hours]: This experiment generates the results of Table 7 in the paper.*
Execution: Execute `script/minigpt4/minigpt4_coco_script/script_sentiment_minigpt4_defense.sh` to apply Gaussian and JPEG defense during inference.
Results: Evaluate the results using `eval_instruction_following.ipynb` with correct file paths.

(E7): *[Anomaly detection defense] [1 human-hour + 1.5 compute-hours]: This experiment generates the results of Table 8 in the paper.*
Execution: Run the notebook `eval_anomaly_detection.ipynb` to generate augmentations and compare similarity between image soft prompt embeddings and their augmentation embeddings with unperturbed image embeddings and their augmentation embeddings.
Results: The anomaly detection results will be displayed in the notebook.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: TORCHLIGHT: Shedding LIGHT on Real-World Attacks on Cloudless IoT Devices Concealed within the Tor Network

Yumingzhi Pan[†], Zhen Ling[†,*], Yue Zhang[‡], Hongze Wang[†], Guangchi Liu[†], Junzhou Luo[†], Xinwen Fu[§]
[†]Southeast University, Email: {pymz, zhenling, wanghongze, gc-liu, jluo}@seu.edu.cn
[‡]Drexel University, Email: yz899@drexel.edu
[§]University of Massachusetts Lowell, Email: xinwen_fu@uml.edu

A Artifact Appendix

A.1 Abstract

This artifact is the public source release of TORCHLIGHT which is designed to detect both known and unknown threats targeting cloudless IoT devices by analyzing Tor traffic. TORCHLIGHT consists of three components:

- Tor Exit Traffic Collector: Real-time capture and storage of external traffic on a resource-constrained VPS, with offline filtering on a campus NAS. *Artifacts released:* node deployment guidelines, offline traffic filtering code.
- Deployment Planner: Optimized VPS resource allocation for effective traffic monitoring. *Artifacts released:* Tor weighted bandwidth algorithm, Tor exit deployment strategy code.
- LLM-based IoT Traffic Analyzer: IoT traffic identification and detection of four attack types using a Large Language Model. *Artifacts released:* IoT traffic identification and attack detection code, sample data.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Executing this artifact does not involve security, privacy, or ethical concerns. For ethical reasons, we will not make the Tor exit traffic we collected publicly available. However, in order to help evaluators in testing the IoT traffic identification and attack detection functionalities, we have released a portion of sample data in accordance with the Menlo Report:

1. This data set excludes any information that could potentially reveal personal or identifiable details—including the Host field, cookies, and software version fingerprints.
2. For attack detection sample data, these malicious requests solely exploit N-day vulnerabilities that have already been patched.

Please refer to these samples in the `./sample_data` directory in the artifact.

Furthermore, to facilitate researchers in validating the functionality of the data preprocessing scripts, we have downloaded several test PCAP files from publicly accessible websites and repositories (including SampleCaptures and PCAPS). Please refer to the `./test_pcaps` folder to access the test PCAP files.

A.2.2 How to access

The code, Tor exits deployment guidelines, and test data are available on Zenodo: https://zenodo.org/records/14742809.

A.2.3 Hardware dependencies

GPU. For our LLM-based IoT traffic analyzer, GPUs with at least 48GB of memory (such as NVIDIA A40, A6000, and A100) are required to meet the memory demands of the quantized Llama 70B model.

A.2.4 Software dependencies

Python Package. All necessary Python packages for our LLM-based IoT traffic analyzer are listed in the corresponding `requirements_<dir>.txt` files.

- **IoT Traffic Identification**
  - **Python version**: Python 3.9.17
  - Install dependencies in the `./iot_identification` directory
  - Install and extract the exllama repository. After extraction:

[*]Corresponding author: Prof. Zhen Ling of Southeast University, China.

    - Copy these Python files to `iot_identification` folder: `model.py`, `generator.py`, `cuda_ext.py`, `lora.py`, and `tokenizer.py`
    - Copy the entire `exllama_ext` directory

- **Attack Detection**
  - **Python version**: Python 3.9.19
  - Install dependencies in the `./attack_detection` directory
- **Preprocessing**
  - Install dependencies in the `./preprocessing` directory

Environment Recommendation Due to version discrepancies in Python (3.9.17 vs 3.9.19) and potential dependency conflicts between components, we strongly recommend using separate virtual environments through Conda environments or Python virtualenv.

Open-Source LLM Models. For IoT Traffic Identification, it is necessary to download a quantized Llama 2 70B model(released at `https://huggingface.co/TheBloke/Llama-2-70B-Chat-GPTQ`). For Attack Detection, a quantized Llama 3.1 70B model is required for download (released at `https://huggingface.co/hugging-quants/Meta-Llama-3.1-70B-Instruct-AWQ-INT4`).

A.2.5 Benchmarks

Tor Exit Sample Data. Due to the sensitivity of Tor exit traffic, the raw data we collected will not be made publicly available. However, in order to help evaluators in testing the IoT traffic identification and attack detection functionalities, we have released a portion of sample data in accordance with the Menlo Report. Please refer to these samples in the `./sample_data` directory in the artifact.

Test Pcaps. To facilitate researchers in validating the functionality of the data preprocessing scripts, we have downloaded several test PCAP files from publicly accessible websites and repositories (including SampleCaptures and PCAPS). Please refer to the `./test_pcaps` folder to access the test PCAP files.

A.3 Set-up

A.3.1 Installation

The installation steps are explained in detail in the `README.md` file. Specifically,

Tor Exit Traffic Collector. For detailed instructions on deploying Tor exits and collecting Tor external traffic data, please refer to `deployment_guideline.md` in `./deployment` directory.

Deployment Planner. Download the latest Tor consensus file. One optional source for downloading is `https://torstatus.rueckgr.at/`, which converts the consensus file into a CSV format, making it easier to process.

LLM-based IoT Traffic Analyzer. Install the dependencies using the `requirements_<dir>.txt` files in the `./iot_identification` and `./attack_detection` directories.

A.3.2 Basic Test

The execution steps are explained in detail in the `README.md` file.

Tor Exit Traffic Collector. Users can run Suricata, Tor, lsyncd, iptables, and ipset to test if the software components are functioning fine.

Deployment Planner. Users can execute `calculate_weights.py` to verify the correct download of the consensus file.

LLM-based IoT Traffic Analyzer. Users can run the following commands to see if the Python packages have been installed successfully.

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer,
↪ AwqConfig
import pandas as pd
from model import ExLlama, ExLlamaCache, ExLlamaConfig
from tokenizer import ExLlamaTokenizer
from generator import ExLlamaGenerator
import httplib2
from googleapiclient.discovery import build
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Tor Exit Traffic Collector captures and storages of external traffic in real-time on a resource-constrained VPS. (§6.2 "Traffic Distribution")

(C2): Deployment Planner optimizes VPS resource allocation for effective traffic monitoring. (§6.2 "Traffic Distribution", §6.2 "Effectiveness" and Figure 4)

(C3): LLM-based IoT Traffic Analyzer leverages a LLM to determine if the response data originates from IoT devices, thereby identifying IoT traffic. (§6.2 "Accuracy", §6.3 and Table 3, 4)

(C3): LLM-based IoT Traffic Analyzer prompts the LLM to detect four types of attacks. (Appendix A.2 "Performance", §6.4 and Table 5, 7)

A.4.2 Experiments

Please refer to the `README.md` file in the repository for detailed steps of our experiments.

(E1): *[Tor Exit Traffic Collector] [30 human-minute + 10 compute-minute + 2GB disk]:*

Preparation & Execution: Please refer to the detailed instructions on deploying Tor exits and collecting Tor external traffic in `deployment_guideline.md` in `./deployment` directory. For offline filtering, please refer to "Attack Detection Preprocessing" in the `README.md` file.

Results: Tor Exit Traffic Collector shoud be able to collect Tor external traffic and preprocess the traffic into appropriate formats for LLMs.

(E2): *[Deployment Planner] [10 human-minute + 2 compute-minute]:*

Preparation: Download the latest Tor consensus file as detailed in A.3.1.

Execution: To compute the bandwidths E and D, and determines weights Wee and Wed based on three distinct network conditions, please execute `deployment/calculate_weights.py`. Then, execute `deployment/deploy_strategy.py` to formulate the deployment plan (There are example node options included in the script).

Results: Deployment Planner shoud be able to generate a deployment plan that optimizes VPS resource allocation for effective traffic monitoring.

(E3): *[IoT Traffic Identification] [2 human-hour + 10 compute-minute + 50GB disk]:*

Preparation: (i) Activate the IoT Traffic Identification virtual environment as specified in the A.2.4 section. (ii) Obtain an API key and search engine ID for Google Custom Search Engine(https://programmablesearchengine.google.com). (iii) Utilize the data provided in the `iot_identification_sample_data.csv` file in `./sample_data` directory.

Execution: After entering `iot_identification` directory, run the following four sequential steps:

- **Step I**: It preliminarily identify IoT names within the response data. (`01a`, `01b`)

```
python 01a_llama_ner_initial.py
python 01b_extract_result_from_initial.py
```

- **Step II**: It re-verifies the IoT entities to address potential hallucinations. (`02a`, `02b`)

```
python 02a_llama_ner_second_round.py
python 02b_extract_result_from_second_round.py
```

- **Step III**: It leverages a search-engine-based retriever to complete potentially missing vendor and type names based on the identified model names. (`03a`, `03b`, `03c`)

```
python 03a_retrieve_google_search.py
python 03b_llama_ner_google_search.py
python 03c_extract_result_from_google_search_llamaed.py
```

- **Step IV**: It ensures that the traffic truly originates from IoT devices. (`04a`, `04b`)

```
python 04a_llama_ner_device_traffic_check.py
python 04b_extract_result_from_device_traffic_check.py
```

Results: Upon successful execution of Step IV, the final `<OUTPUT_FILE>` will contain LLM-identified IoT traffic entries, along with their corresponding predictions for vendor (`pred_vendor`), device type (`pred_type`), and/or model (`pred_model`).

(E4): *[IoT Attack Detection] [2 human-hour + 10 compute-minute + 50GB disk]:*

Preparation: (i) Activate the IoT Traffic Identification virtual environment as specified in the A.2.4 section. (ii)Utilize the data provided in the `attack_detection_sample_data.csv` file in `./sample_data` directory.

Execution:

- **Command injection detection.**

```
python llama_command_injection.py
python extract_llama_command_injection.py
```

- **FTP anomaly detection.**

```
python llama_ftp_anomaly.py
python extract_llama_ftp_anomaly.py
```

- **Path traversal detection.**

```
python llama_path_traversal.py
python extract_llama_path_traversal.py
```

- **Information disclosure detection.**

```
python llama_information_disclosure.py
python extract_llama_information_disclosure.py
```

Results: (i) The output file from `llama_<attack_type>.py` contains the LLM's complete response to this specific attack type. This includes whether the LLM identified the input as an attack and its subsequent analysis, recorded in the `<attack_type>_pred_line` column. (ii) The `extract_llama_<attack_type>.py` script extracts the LLM's core answer (`"yes"` or `"no"`) regarding the attack and stores it in the `<attack_type>_pred_result` column.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: CloudFlow: Identifying Security-sensitive Data Flows in Serverless Applications

Giuseppe Raffa
Royal Holloway University

Jorge Blasco
Universidad Politécnica

Dan O'Keeffe
Royal Holloway University

Santanu Kumar Dash
University of Surrey

A Artifact Appendix

This artifact appendix is dedicated to `CloudFlow` and `CloudBench`. The first is a serverless-specific static analysis framework, whilst the second is a suite of microbenchmarks used for the evaluation of `CloudFlow`.

A.1 Abstract

In our paper, we present `CloudFlow`, a novel framework to statically detect security-sensitive data flows in serverless applications. To achieve this, `CloudFlow` leverages the infrastructure definition provided by the developer to identify the events, permissions and entry points of an application. We evaluate our framework against a new suite of 40 microbenchmarks, `CloudBench`. Furthermore, we analyse 104 real-world applications selected from AWSomePy, a recent dataset. To the best of our knowledge, this is the largest security-focused analysis of serverless applications to date. Our results show that `CloudFlow` passes all microbenchmarks, apart from three, and detects 11 code injection and information leakage vulnerabilities in the analysed real-world applications. To support the verification of the functionality of our framework and the reproducibility of our results, we open-source both `CloudFlow` and `CloudBench`. In addition, we provide a ready-to-use environment that facilitates their execution. Finally, we make available for future reference a report detailing the security-sensitive data flows identified in AWSomePy.

A.2 Description & Requirements

The artifact presented in this appendix consists of two components. The first is a Linux Ubuntu 20.04 Virtual Machine (VM) that contains the source code of `CloudFlow` and `CloudBench` as well as a pre-processed version of AWSomePy. The VM also includes the installation of the `CloudFlow` command-line utility and Pysa, the underlying static analysis tool used in our framework. The second component is the data flow report of the AWSomePy dataset analysis. This includes the information obtained from `CloudFlow` and our observations on the manually assessed security-sensitive data flows.

A.2.1 Security, privacy, and ethical concerns

The usage of `CloudFlow` and `CloudBench` does not involve taking any destructive steps. With regard to the vulnerabilities discussed in §5.2 and §5.3 of our paper, they were responsibly disclosed in November 2024. More information on this topic can be found in the Ethics Considerations section of the paper.

A.2.2 How to access

Our artifact is available on Zenodo at the following address:

- `https://doi.org/10.5281/zenodo.15609299`

The provided VM is password-protected, and it can be accessed as follows:

- User: Giuseppe Raffa
- Password: USENIX2025

We recommend importing the VM in a recent version of VirtualBox™. If the Ubuntu desktop does not appear automatically after logging in, we suggest adjusting the window size from the main VirtualBox interface by clicking on *View → Adjust Window Size*.

A.2.3 Hardware dependencies

No special hardware is required to run `CloudFlow` and `CloudBench`. However, we recommend allocating 8GB of RAM to the VM. This value is already configured in the provided `.ova` file.

A.2.4 Software dependencies

When we exported our VM, we chose *Open Virtualization Format 2.0*, as this was the most recent version of the standard supported by our hypervisor. While we are not aware of any limitations relating to the Open Virtualization Format, as

previously mentioned, we recommend importing the VM in a recent version of VirtualBox.

A.2.5 Benchmarks

`CloudBench` is the only relevant collection of benchmarks. We note that `CloudBench` is included in our VM.

A.3 Set-up

`CloudFlow` was implemented as a command-line (CLI) utility that runs within a dedicated Python Virtual Environment (VE). The latter is available within the folder `pysa_experiments` located on the `Desktop` of the VM. As detailed in §A.3.2, the VE **must be activated** before launching the utility. Also, we recommend deactivating the VE by executing the command *deactivate* in the console prior to switching off the VM.

A.3.1 Installation

The shared execution environment is a fully-fledged, ready-to-use VM. Therefore, no installation is required. The above-mentioned `pysa_experiments` folder contains the required source code within the following folders[1]:

- `awsomepy-serverless`: Pre-processed dataset.
- `cloudbench`: Microbenchmarks suite.
- `cloudflow`: Framework source code[2].

The folder `pysa_experiments` also contains two VEs[3]:

- `experiments`: Main `CloudFlow` VE. This includes Pyre version **0.9.10** (package `pyre-check`).
- `venv-sapp`: Auxiliary VE for report generation. This includes SAPP version **0.5.2** (package `fb-sapp`).

To allow future modifications of our artifact, we note that our VM includes on its `Desktop` a folder called `pyre-check`. It contains a collection of resources obtained from the Pyre GitHub repository[4]. This folder is part of our environment because we were unable to install and configure Pyre in such a way that those resources were automatically detected. We emphasize that `CloudFlow` does **not** work without that folder.

[1] To improve their accessibility, our source code and dataset are also included in a ZIP archive stored in Version v1 of the Zenodo record.

[2] Note that all the `CloudFlow` configuration files are stored within the folder `pysa_experiments/cloudflow/cloudflow/config`.

[3] `CloudFlow` activates the auxiliary VE to use the APIs of the SAPP reporting tool. We made this choice due to installation problems experienced while trying to make Pyre and SAPP available within the same VE.

[4] https://github.com/facebook/pyre-check

A.3.2 Basic Test

The objective of the basic functionality test described in this section is to check that `CloudFlow` can be successfully executed. The test consists of 4 steps:

(S1): VE Activation. Open a CLI window and select the `pysa_experiments` folder as the current folder. Activate the `CloudFlow` VE by running the following command: *source experiments/bin/activate*
Expected Output: The CLI prompt will change, and it will include the VE name, i.e., `experiments`, in parenthesis.

(S2): VE Content. The content of the VE can be inspected by executing the following command: *pip list*
Expected Output: `CloudFlow` is present in the list of the packages available in the VE. Specifically, the information associated to `CloudFlow` is the version 0.0.1 and its local installation path.

(S3): Unit Tests Execution. Our framework includes unit tests that can be run to exercise most of its modules. After selecting `pysa_experiments/cloudflow` as current folder, the unit tests can be launched with the following command: *pytest -k "not microbenchmarks"*
Expected Output: The command executes 79 unit tests. All of them are expected to pass.

(S4): Single Microbenchmark Execution. With the same current folder as the previous step, the `CloudBench` microbenchmark `owasp-serverless-injection` can be run by launching `CloudFlow` in single repository mode, i.e., with the *-s* CLI flag[5].
Expected Output: The framework identifies a security-sensitive data flow between line 12 and line 26 of the file `httphandler.py`. The result is displayed in an on-screen table. Importantly, `CloudFlow` supports multiple CLI flags. To access their documentation, execute the command: *cloudflow -h*

A.4 Evaluation workflow

In this section, we first summarize the major claims made in our paper (§A.4.1). Next, we describe a set of experiments (§A.4.2). Each of these is focused on a specific claim.

A.4.1 Major Claims

In what follows, we list the major claims of our paper:

(C1): `CloudFlow` passes all the microbenchmarks in the `CloudBench` suite, apart from three. The results of our microbenchmark-based evaluation are reported in §4.2 of our paper.

[5] *cloudflow -s /home/giuseppe/Desktop/pysa_experiments/cloudbench/intra-procedural/s3-service/owasp-serverless-injection*

(C2): `CloudFlow` detects 205 security-sensitive data flows in 104 AWSomePy applications. We analyse such data flows in §5.2 of our paper. We note that the AWSomePy dataset contains 145 applications in total. However, we focus our attention on 104 of them after an initial triage, summarized in Table 3 of the paper.

(C3): `CloudFlow` identifies 11 vulnerabilities in the analysed AWSomePy applications. Such vulnerabilities are summarized in Table 5 of our paper. The latter also includes two vulnerability case studies in §5.3.

A.4.2 Experiments

In this section, we describe the proposed experiments.

(E1): Microbenchmark-based Evaluation. We estimate: 2 compute-hours + 5 human-minutes. This experiment is designed to validate our claim C1.

Description: `CloudFlow` is launched in microbenchmark mode, i.e., with a CLI flag that allows executing all the `CloudBench` microbenchmarks. The analysis of the results is performed by running a PyTest module.

Preparation: Ensure that the `CloudFlow` VE is active (§A.3.2) and that `pysa_experiments/cloudflow` is the current folder.

Execution: Two steps are required. First, launch the execution of all `CloudBench` microbenchmarks with the command: *cloudflow -mb all*. Second, when the execution of the microbenchmarks is complete, run the relevant PyTest module as follows: *pytest -k microbenchmarks*

Results: Check that the PyTest on-screen summary reports that three microbenchmarks out of 40 have failed. The names of the failed microbenchmarks are specified in §4.2 of our paper[6].

(E2): Security-sensitive Data Flows in AWSomePy. We estimate: 5 compute-hours + 30 human-minutes. This experiment is designed to validate our claim C2.

Description: `CloudFlow` is launched in multiple repository mode, i.e., with the *-m* CLI flag, and generates a report. The latter needs to be compared with the report available on Zenodo.

Preparation: Same as the previous experiment.

Execution: Launch the analysis of the AWSomePy dataset[7]. Unlike the previous experiment, `CloudFlow` is launched with the additional *-cf* flag, used to specify a required configuration file. At the end of the execution, the framework generates a data flow report within the folder `pysa_experiments/cloudflow-report-files`.

Results: To correctly compare the report generated by this experiment with that available on Zenodo, we observe that the latter contains:

- A total of 222 data flows. These include those detected in applications filtered out during our dataset triage as well as others categorized as invalid. The 205 data flows mentioned in claim C2 are those classified as **Valid** in the column Assessment. We expect that the execution of this experiment will identify the same 222 data flows, as the folder `awsomepy-serverless` (§A.3.1) includes all the AWSomePy applications for completeness.
- Six columns **not** generated by `CloudFlow` (from Assessment to Comments). We make these columns publicly available to better document our analysis.

(E3): Vulnerabilities in AWSomePy. We estimate: 3.5 human-hours[8]. This experiment is designed to validate our claim C3.

Description: This experiment requires manually validating at least a sample of the 11 security-sensitive data flows classified as vulnerabilities in the report available on Zenodo. These are marked with the value **Yes** in the column Security Vulnerability.

Preparation: Download the report in CSV format available on Zenodo. This includes all the required information, e.g., the location of sources and sinks.

Execution: Analysis of at least a sample of the security-sensitive data flows classified as vulnerabilities. With regard to the case studies discussed in §5.3 of the paper, we observe that[9]:

- The Code Injection via HTTP case study refers to the vulnerabilities found in the application with identifier 040AA.
- The Exception Traceback Leakage case study refers to the vulnerabilities found in the application with identifier 022AA.

Results: We expect the manual validation to confirm the identified vulnerabilities. We note that the vulnerabilities found in the applications 040AA and 140AA are associated with data flows featuring the same source and sink. To the best of our knowledge, they are reported twice by Pysa because two data flows connecting the source with the sink are detected. Therefore, we consider these as separate data flows.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

[6] For a shorter on-screen report: *pytest -k microbenchmarks --tb=no*

[7] *cloudflow -m /home/giuseppe/Desktop/pysa_experiments/awsomepy-serverless/repositories -cf awsomepy_dataset_config_file.yml*

[8] Estimate based on the assumption that the validation of a data flow takes, on average, 20 minutes, and that all the relevant data flows are validated.

[9] Note that the code listings provided in the paper to illustrate our case studies are simplified versions of the actual application code.



USENIX Security '25 Artifact Appendix: Serverless Functions Made Confidential and Efficient with Split Containers

Jiacheng Shi, Jinyu Gu, Yubin Xia, Haibo Chen
Institute of Parallel and Distributed Systems (IPADS), SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

A Artifact Appendix

A.1 Abstract

The artifact includes the following components: (1) the source code of CoFunc system, including the CVM OS code (`cvm_os`), the shadow container code (`shadow_container`) and the patches for the host Linux/QEMU (`patches`); (2) the serverless functions utilized for the evaluation (`testcases/testcases`); (3) the scripts for conducting the experiments (`scripts` and `testcases/tools`). Users of this artifact can evaluate the performance of serverless functions under different container runtimes, including split containers (CoFunc), CVM-based Kata Containers (Kata-CVM), and native lean containers (Native).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact poses no harm to machine security or data privacy.

A.2.2 How to access

The artifact is available at Github (https://github.com/shijc-sjtu/cofunc-artifact) and Figshare (https://doi.org/10.6084/m9.figshare.28234346.v4).

A.2.3 Hardware dependencies

The artifact requires AMD CPUs with SEV-SNP support. It has been tested on an EPYC-7T83 machine. At least 96 CPU cores and 180GB of memory are required.

A.2.4 Software dependencies

The artifact works on an SEV-SNP version of the Linux kernel (https://github.com/AMDESE/linux.git, branch `svsm-preview-hv-v2`), along with the modifications in `patches/linux.patch`. The following dependencies are required for building the artifact and running the experiments: Docker, screen, Python 3 (with matplotlib, numpy, boto3, pandas, CouchDB) and gcc.

A.2.5 Benchmarks

The Dockerfiles for building the serverless functions are available at `testcases/testcases`.

A.3 Set-up

A.3.1 Installation

For reviewers using the test server, the artifact is already installed, and no further steps are required. Other users need to build the artifact components and set up the host environment with the steps described in the README.

A.3.2 Basic Test

Run the script `scripts/run_simple.sh`, which executes a serverless function using CoFunc. The script will output the function's running time.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): For the 28 evaluated functions, CoFunc demonstrates significant performance improvements (up to 60×) compared with Kata-CVM, while incurring <14% performance overhead compared with Native. This is proven by the experiment (E1) described in Section 7.1 whose results are illustrated in Figure 11.

(C2): CoFunc outperforms Kata-CVM on FINRA application by 31× when 200 auditing functions start concurrently. This is proven by the experiment (E2) described in Section 7.4 whose results are reported in Section 7.4.

A.4.2 Experiments

(E1): [1.5 compute-hours]: Evaluate the end-to-end latencies (handling a single request) of the functions using CoFunc, Kata-CVM, and Native.
Preparation: None.
Execution: Run the script `scripts/run_fig11.sh`. This script executes the functions with different runtimes and outputs the latencies to the `log` directory.

Results: The script will generate a table at `plots/fig11.txt` that contains the function latencies and the overhead/optimization of CoFunc compared with Native/Kata-CVM. Additionally, the script will generate a figure at `plots/fig11.pdf`, which can be compared with Figure 11.

(E2): [10 compute-minutes]: Evaluate the end-to-end latency of FINRA application with 200 concurrent auditing functions using CoFunc and Kata-CVM.
Preparation: None.
Execution: Run the script `scripts/run_finra.sh`. This script executes FINRA application with different runtimes and outputs the latencies to the `log` directory.
Results: The end-to-end application latencies and the optimization of CoFunc compared with Kata-CVM can be found in `plots/finra.txt`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Exploring and Exploiting the Resource Isolation Attack Surface of WebAssembly Containers

Zhaofeng Yu[1], Dongyang Zhan[1]*, Lin Ye[1], Haining Yu[1], Hongli Zhang[1], Zhihong Tian[2]
[1]Harbin Institute of Technology, [2]Guangzhou University
yuzhaofeng@stu.hit.edu.cn
{zhandy, hityelin, yuhaining, zhanghongli}@hit.edu.cn
tianzhihong@gzhu.edu.cn

A Artifact Appendix

A.1 Abstract

This artifact provides a comprehensive research toolset for analyzing WebAssembly runtimes and evaluating the resource exhaustion exploits proposed in our paper. It consists of two major components: a static analysis framework and a suite of automated exploit scripts.

The static analysis framework performs static analysis of WASI/WASIX interfaces to identify external API calls and potential syscall invocations, along with their relevant parameters. To support reproducibility, we provide both a Docker image with all dependencies pre-installed and standalone automation scripts for non-containerized environments.

The exploit part of the artifact demonstrates how crafted WebAssembly instances can exhaust host resources via legitimate runtime interfaces. It includes test cases and automation scripts to reproduce our evaluation results and measure the impact of each exploit instruction.

A README file is provided, which contains detailed setup and usage instructions. This artifact is released for academic use only. Any malicious application is strictly prohibited.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The static analysis component of our artifact does not pose any security risks. During exploit evaluation, some test cases may temporarily affect system performance, such as I/O throughput and network bandwidth. For network-related experiments, users need to specify an IP and port, and run tools like `ncat` and `iperf3` on a separate host to receive traffic (see README for details). Our artifact does not modify or collect any user data. It does not perform any destructive actions.

* Corresponding author: Dongyang Zhan (Email: zhandy@hit.edu.cn)

A.2.2 How to access

Our artifact is publicly available on Zenodo. Please visit the following DOI to access the latest version: https://doi.org/10.5281/zenodo.16594863.

The archive contains all necessary files, including analysis tools, exploit scripts, Docker images, and documentation. Please refer to the included `README` file for detailed setup and usage instructions.

A.2.3 Hardware dependencies

The artifact does not require any specialized hardware. However, we recommend a system with at least 16 GB of RAM (ideally 32GB) and a reasonably powerful CPU to ensure smooth analysis and reasonable execution time.

A.2.4 Software dependencies

To begin with, we strongly recommend reading the provided `README` file, which offers detailed instructions for environment setup and dependency installation. As an initial step, please ensure that `p7zip-full` is installed in order to extract the artifact package.

If using the Docker image, the analysis framework requires only Docker to be installed. However, for users who prefer to run the framework locally without Docker, it is necessary to manually install LLVM (version 17.x) and a nightly version of Rust that is compatible with the selected LLVM version. In addition, several other system dependencies are required, including `curl`, `cmake`, `gcc`, `g++`, `python3`, `python3-dev`, `zlib1g`, `make`, `binutils`, `bzip2`, `zip`, `xz-utils`, and `git`.

For evaluating the exploits, a separate set of dependencies must be installed. These include `python3`, `python3-pip`, `dstat`, `ent`, `fio`, `iperf3`, and `sysstat`. Additionally, the Python package `pandas` is required and can be installed using `pip`.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

For cases where the provided Docker image is used to test the static analysis framework, please follow the steps described in Listing 1.

```
# Step 1: Install Docker
# Step 2: Unpack the archive to retrieve our
    provided Docker image.
sudo apt update
sudo apt install p7zip-full
7z x artifact.image.7z.001
# Step 3: Load the Docker image
sudo docker load -i artifact.image
```

Listing 1: Environment setup for evaluating the static analysis framework with Docker.

For cases where the static analysis framework is tested in a non-containerized environment, please follow the steps described in Listing 2.

```
# Step 1: Install dependencies
sudo apt --yes update && sudo apt install --yes
    cmake gcc g++ \
              python3 python3-dev zlib1g make
                  binutils bzip2 \
              zip xz-utils git
git clone https://github.com/llvm/llvm-project.git
mkdir ./build && cd ./build/
cmake -S ../llvm-project/llvm/ -B ./ -G "Unix␣
    Makefiles" \
              -DCMAKE_BUILD_TYPE=Release \
              -DLLVM_ENABLE_PROJECTS="clang;clang-
                  tools-extra;lldb;lld" \
              -DLLVM_ENABLE_RUNTIMES="libcxx;
                  libcxxabi;compiler-rt;libunwind" \
make -j 8
sudo make install
sudo apt install --yes curl
curl --proto '=https' --tlsv1.2 -sSf https://sh.
    rustup.rs | sh
rustup toolchain install nightly-2024-02-01
rustup default nightly-2024-02-01-x86_64-unknown-
    linux-gnu
rustup component add rust-src --toolchain nightly
    -2024-02-01-x86_64-unknown-linux-gnu
# Step 2: Unpack the archive to retrieve our
    artifact.
sudo apt update
sudo apt install p7zip-full
7z x wasm_artifacts.7z.001
```

Listing 2: Environment setup for evaluating the static analysis framework without Docker.

To evaluate the exploit strategies proposed in our paper, please follow the steps described in Listing 3 to set up the environment.

```
# Step 1: Install dependencies
sudo apt install python3 python3-pip
pip install pandas
curl https://wasmtime.dev/install.sh -sSf | bash
curl https://get.wasmer.io -sSfL | sh
sudo apt install dstat ent fio iperf3 sysstat
# Step 2: Unpack the archive to retrieve our
    artifact.
sudo apt update
sudo apt install p7zip-full
7z x wasm_artifacts.7z.001
```

Listing 3: Environment setup for evaluating the proposed exploit strategies.

A.3.2 Basic Test

To test our static analysis framework, please follow the instructions described in Listing 4 to run it.

```
# After loading the provided Docker image, you can
    run the container to automatically demonstrate
    the workflow of the static analysis framework.
sudo docker run -it --rm artifacts:latest
# After installing the dependencies and extracting
    the provided artifact, navigate to the directory
     and run the automated script.
cd wasm_artifacts
./start.sh
```

Listing 4: Running the static analysis framework via Docker or automated script.

To evaluate the exploit strategies proposed in our paper, please follow the instructions described in Listing 5.

```
# After installing the dependencies and extracting
    the provided artifact, navigate to the directory
     and run the automated script.
cd wasm_artifacts/exploit
./exploit.sh
# Note: If you need to evaluate network-related
    strategies, each time you run the script (
    including the first), you must provide two
    parameters. You can run ncat and iperf3 on
    another machine and then supply their listening
    IP addresses and ports to the script.
./exploit.sh <send_traffic_ip:port> <iperf_server_ip:
    port>
```

Listing 5: Evaluating our proposed exploit strategies via automated script.

A.4 Evaluation workflow

A.4.1 Major Claims

Our artifact consists of two components: the static analysis framework and our proposed exploit strategies. To evaluate the artifact, one can examine the outputs produced by both components, as described below.

(C1): *In experiment (E1), the static analysis framework is executed using the provided Docker image (recommended) or the automated script, producing a JSON file that contains key external APIs or syscalls involved in the runtime interfaces, along with some possible parameters. Table 1 in Section 3.4 of our paper illustrates part of the contents of this JSON file.*

(C2): *In experiment (E2), the provided automated script is used to measure the system's baseline performance and evaluate the impact of each exploit strategy on system performance. The output results are saved in* `wasm_artifacts/exploit/data/result`*. In the WebAssembly container, our strategies can degrade system performance in various aspects, such as I/O, CPU usage, and network bandwidth. Several tables in Section 5 of our paper present these results.*

A.4.2 Experiments

(E1): *[Run the static analysis framework] [About 1 hour.]: By running our static analysis framework via the Docker image or the automated script, the framework will automatically build the project, extract the IR, perform analysis, and finally output a JSON file containing the key external APIs and syscalls involved in the runtime interfaces, along with some possible parameters.* ***Note that this process takes a considerable amount of time. Therefore, we have included a GIF in the README to demonstrate the procedure.***

Preparation: *If you want to run the static analysis framework using the Docker image (recommended), follow the steps in Listing 1. To run it using the automated script, follow the installation steps in Listing 2.*

Execution: *The basic commands are listed in Listing 4, which by default analyze the Wasmtime runtime. The source code of the runtime is already included in both the Docker image and the artifact, so no additional download is required. If you wish to analyze Wasmer instead, an extra parameter must be provided when running the container or script:* `/wasm_artifacts/wasm_runtimes/wasmer/` *for the container, and* `./wasm_runtimes/wasmer/` *for the script. Note that analyzing Wasmer can take significantly more than one hour due to its large codebase. It is also recommended to ensure that the system has more than 30GB of available memory to avoid analysis failures due to memory exhaustion.*

Results: *After the Docker container or automated script finishes running, the contents of the JSON file will be displayed using* `cat`*. In the container, the JSON file is saved at* `/wasm_artifacts/result`*, while the script saves it to* `./wasm_artifacts/result`*.*

(E2): *[Optional Name] [About 4 hours]: By running the provided automated script multiple times, the system baseline performance and the system performance during the execution of the exploit strategies can be collected.* ***We have also included a GIF in the README to demonstrate the procedure.***

Preparation: *To evaluate the impact of the exploit strategies, first install the dependencies by following the instructions in Listing 3.*

Execution: *The basic commands are listed in Listing 5, which will first measure the system baseline performance and then evaluate the impact of the first exploit strategy. By running the script multiple times, you can continuously evaluate the next exploit strategy.*

Results: *When the script is run but produces no output, it means all exploit strategies have been evaluated. All collected data can be found in the* `wasm_artifacts/exploit/data/result` *directory.*

A.5 Notes on Reusability

All scripts and source code for our static analysis framework and exploit tools are included in the artifact. If needed, new features can be added by simply modifying the scripts and source code. Additionally, the analysis-related source code can be directly integrated into your own project and used by calling the provided class interfaces, requiring only a proper LLVM installation.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Transparent Attested DNS for Confidential Computing Services

Antoine Delignat-Lavaud
Azure Research, Microsoft

Cédric Fournet
Azure Research, Microsoft

Kapil Vaswani
Azure Research, Microsoft

Manuel Costa
Azure Research, Microsoft

Sylvan Clebsch
Azure Research, Microsoft

Christoph M. Wintersteiger
Imandra

A Artifact Appendix

A.1 Abstract

Our artifacts consist mainly of an implementation of an authoritative DNSSEC server that implements the attested DNS protocol, as described in the paper, and a browser extension that implements the client protocol.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our implementation of attested DNS is for research purposes only. We do not recommend directly exposing our attested DNS server to the public Internet, as it lacks some of the recommended defenses against classical DNS-based attacks, in particular DNS amplification attacks over UDP.

If you want to test our server over the public Internet (for instance, to test client latency or the browser extension), consider exposing it through Bind using a forwarder zone. We provide a sample Bind configuration, see the README file for details.

A.2.2 How to access

The version of attested DNS used in this paper is available from https://doi.org/10.5281/zenodo.15611255. It is based on the open source repository https://github.com/microsoft/ccfdns - however, please note that the open source repository has significantly diverged from the version described in the paper. While the open source version is significantly more up to date, it is missing several features that are evaluated in the paper.

A.2.3 Hardware dependencies

Our artifacts requires Intel SGX capable hardware and must run with DCAP (Data Center Attestation Primitives) attestation. EPID-based attestation is not supported. We have tested our code on Azure `Standard_DC*ds_v3` VMs, other cloud providers are expected to work but may require additional OpenEnclave configuration steps that we do not provide. Refer to the OpenEnclave manual for details on other platforms (including non-cloud based setups and simulation mode).

A.2.4 Software dependencies

This version of attested DNS has been tested on Ubuntu 20.04. Our main dependencies are OpenEnclave (https://github.com/microsoft/openenclave), and the Confidential Consortium Framework (https://github.com/microsoft/CCF), but our artifact package includes compatible versions. We use DNSPyre (https://github.com/Tantalor93/dnspyre) for benchmarking our server.

A.2.5 Benchmarks

None.

A.3 Set-up

We assume you have downloaded the artifact package and extracted it to your home directory `~/adns`.

A.3.1 Installation

We have automated the installation of dependencies using an Ansible playbook. You can run the playbook with the following commands:

```
cd ~/adns/CCF/getting_started/setup_vm
./run.sh app-dev.yml
```

This script will also install Ansible and its dependencies.

The next step is to install CCF:

```
cd ~/adns/CCF
sudo apt install ./ccf_sgx_5.0.0_dev12_amd64.deb
```

To build the aDNS server, you will also need LLVM 18, which you can install with:

```
sudo apt install clang-18
```

The next step is to build the aDNS enclave:

```
cd ~/adns/ccfdns
./build.sh
```

A.3.2 Basic Test

The main tool for testing our aDNS server is a script that starts an instance of aDNS on port 5353 configured for the zone `service.confidential`. The script also registers two partifipating TEEs under `sgx.service.confidential` and `sev-snp.service.confidential`. To simplify the evaluation and avoid multiple hardware dependencies, these are not live services but use pre-recorded attestation reports.

```
cd ~/adns/scripts
./run.sh
```

The process of starting the server involves the registration of the two TEEs, the DNSSEC signature of the zone, and the issuance of ACME certificates.

```
Testing SGX service registration...
...
SGX service registration completed in 176.34 ms
Testing registration policy update by consortium member
...
New policy set in 249.15 ms
Testing SEV-SNP service registration
...
SEV-SNP service registration completed in 156.80 ms
Testing re-signature of zone
...
Zone re-signature completed in 14.24 ms
Testing ACME certificate issuance through aDNS
ACME certificate returned: {"certificate":
  "-----BEGIN CERTIFICATE-----
 ....-----END CERTIFICATE-----\n"}
Certificate returned in 2.01 ms
```

The server will keep running until interrupted (stop it with `ctrl+c`). Please note that the experiments below require the server to be running. You can only run one single instance of the server at a time.

For testing aDNS clients, note that the aDNS server is itself a registered TEE of the zone, available at `https://ns1.service.confidential:8443`. Note that all ACME certificates use a local root CA in `~/adns/scripts/pebble.pem`. You HTTPS client should be configured to use that root CA when connecting to aDNS. For instance, to see all the records configured in aDNS's zone, run the following command:

```
curl --cacert pebble.pem \
  https://ns1.service.confidential:8443/app/dump
```

We need to install DNSPyre to measure the performance of the aDNS zone:

```
sudo apt install \
  ~/adns/dnspyre_3.5.1_linux_amd64.deb
```

Next, we setup a local installation of the `bind` DNS server. This will be used later to measure the scalability of aDNS through Bind9, using its highly optimized cache.

```
cd ~/adns
sudo apt install bind9
sudo cp named.conf.* /etc/bind/
sudo service named restart
```

The packaged Bind configuration file will forward queries to `service.confidential` to port 5353. You can check your setup is correct with dig:

```
dig @127.0.0.1 _8443._tcp.ns1.service.confidential TLSA
```

If aDNS is running and well configured you should see the DANE record:

```
;; ANSWER SECTION:
_8443._tcp.ns1.service.confidential. 1440 IN TLSA
    3 1 0 3076301006072A8648CE3D020106052B81...
```

Finally, to help test legacy client, we recommend you set the local bind as your system resolver.

```
sudo cp resolved.conf /etc/systemd/
sudo service systemd-resolved restart
```

We will use the Firefox extension to measure client latency using the browser networking profiler. If you have configured bind as your system resolver, you can connect to `service.confidential` directly, but if you plan to connect from a remote client, you may need to setup a DoH resolver with a custom certificate, which is not recommended to do on a personal device.

```
sudo apt install tightvncserver firefox
```

To setup the extension, we recommend `web-ext`, which is available on the Node Package Manager. Make sure Firefox and Node.js are installed, for instance on Debian systems:

```
curl -fsSL https://deb.nodesource.com/setup_23.x \
  -o nodesource_setup.sh
sudo -E bash nodesource_setup.sh
sudo apt-get install -y nodejs
sudo npm install -g web-ext
```

Then from the extension directory, call `web-ext` to start a browser session with the extension installed:

```
cd adns/extension
web-ext run
```

A.4 Evaluation workflow

A.4.1 Major Claims

We make two main claims in the paper

(C1): The impact of aDNS on the DNS serving infrastructure is at worst 50%, that is, going from clients that only check the A record of a service to clients that verify DNSSEC and request the full attestation evidence from DNS (worst case scenario) only impacts the throughput of the authoritative server by about half.

(C2): aDNS clients that verify attestation at the same time as the TLS handshake do not observe any additional latency for typical client roundtrip times of about 25ms.

A.4.2 Experiments

(E1): *[Server scalability] [40 human-minutes]* We evaluate the latency/throughput by submitting an increasing amount of requests with dnspyre and measuring the P99 latency.

Preparation: Start the aDNS server with `scripts/run.sh` and make sure your local bind server is well configured.

Execution: We use the script `scripts/latency.sh`, which accepts 4 arguments: the type of client to simulate (can be `basic`, `dane` or `adns`), the DNS server IP (`127.0.0.1` by default), its port (`5353` by default) and the number of queries in each measurement (by default 10000). For each 3 type of client, we run the script and collect the reported `Questions per second` and `p99` latency results. We first do this by accessing our aDNS server directly (on port 5353), then through bind (on port 53):

```
./latency.sh basic
./latency.sh dane
./latency.sh adns
./latency.sh basic 127.0.0.1 53
./latency.sh dane 127.0.0.1 53
./latency.sh adns 127.0.0.1 53
```

Results: The results should roughly match the results of Figure 4 and Figure 5. As we increase the number of concurrent requests, the throughput and P99 latency gradually increase. Bind is about 2 orders of magnitude better than our own implementation. More importantly, the maximum throughput achieved by aDNS clients is about half that of basic clients.

(E2): *[Client latency] [5 human-minutes]*

Preparation: Start the aDNS server with `scripts/run.sh`. To keep things simple, we assume you will run Firefox on a machine configured to use the locally configured bind as a system resolver. For instance, use VNC to connect to the server where aDNS is running and launch a Firefox session with the extension using `web-ext run`.

Execution: Start the network profiler by going to the Network tab of the developer tools (F12). Click the stopwatch in the bottom left corner to start profiling. Navigate to `https://ns1.service.confidential:8443` and measure the total latency (until the HTTP request is sent). Disable the extension and repeat the experiment. You can check the extension developer console to see the details of the attestation verification happening in the background.

Results: Unless you have very low RTT to the aDNS server, the total latency should be the same with or without aDNS.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2025/`.



USENIX Security '25 Artifact Appendix: TLBlur: Compiler-Assisted Automated Hardening against Controlled Channels on Off-the-Shelf Intel SGX Platforms

Daan Vanoverloop[1], Andrés Sánchez*[2,4], Flavio Toffalini[2,3], Frank Piessens[1], Mathias Payer[2], and Jo Van Bulck[1]

[1]*DistriNet, KU Leuven,* [2]*EPFL,* [3]*RUB,* [4]*Amazon*

A Artifact Appendix

A.1 Abstract

We provide the automated mitigation pipeline, including LLVM and BOLT instrumentation passes, a modified Intel SGX SDK that includes our custom prefetcher, as well as the benchmark applications to evaluate performance overhead. To assess leakage in practice, we furthermore provide an independent reproduction of the controlled-channel attack on libjpeg and a profiler tool to seamlessly extract page-access traces from victim enclaves.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact includes attacks on real-world Intel SGX processors, which require the installation of the privileged SGX-Step kernel module. We recommend that evaluators run all code on an isolated test machine.

The attack and mitigation code in this artifact is intended solely for reproducing our results. Any use of these results should be conducted responsibly. The mitigation is a research prototype, and we do not recommend its deployment in production environments as-is.

A.2.2 How to access

The artifact files are available on Zenodo (https://doi.org/10.5281/zenodo.15194120) and GitHub (https://github.com/TLBlur-SGX/tlblur/tree/usenix-artifact).

A.2.3 Hardware dependencies

Building and running TLBlur requires hardware with support for Intel SGX2 and AEX-Notify.

*All the contributions of this author were made prior to joining Amazon.

A.2.4 Software dependencies

- A Linux distribution supported by the Intel SGX SDK. We recommend using Ubuntu 22.04.
- C/C++ compiler (`gcc` or `clang`) and linker (`lld` strongly recommended)
- Build tools: `make`, `cmake`, `meson` and `ninja`.
- Stable Rust toolchain (1.76 or later)

A.2.5 Benchmarks

No external benchmarks were used for our evaluation.

A.3 Set-up

A.3.1 Installation

Download artifact. Clone the repository with all submodules.

```
$ git clone --recurse-submodules \
    --branch usenix-artifact \
    https://github.com/TLBlur-SGX/tlblur.git
```

Build. Run `build.sh` to build and install LLVM with TLBlur instrumentation passes, the Intel SGX SDK with TLBlur prefetcher, SGX-Step, benchmark enclaves and tools.

Load SGX-Step. Load the SGX-Step kernel module by running `make clean load` in `sgx-step/kernel`.

A.3.2 Basic Test

1. Run the test program with an enclave binary, e.g. `./install/bin/test-rsa ./install/lib/s-encl-rsa-instrumented-relocs-bolt-opt.so`. Output should contain the following, indicating that the attack succeeded:

```
[../enclaves/rsa/main.c] secure enclave
    ↪ encrypted '1234' to '21921'; decrypted
    ↪ '1234'
[../enclaves/rsa/main.c] --> RECONSTRUCTED KEY
    ↪  '20771' (0x5123)
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): A practical profiler (§7.2) to extract enclave page-access traces (Figures 5, 9 and 10).

(C2): A reproduction of the seminal controlled-channel attack by Xu et al. on libjpeg, and demonstrate that it still leaks with AEX-Notify single-stepping defense (Figures 1 and 8).

(C3): An automated compiler pipeline (§6) that reduces information leakage observable to controlled-channel adversaries by "blurring" page-access traces (§9.1).

(C4): An average slowdown of $\times 2.24$ from instrumentation across a variety of benchmark programs (§9.2.1, §9.2.3, Figure 6 and Table 3).

(C5): An interrupt-resumption overhead that scales linearly with the size of the PWS and the size of the enclave (§9.2.2, Figure 7).

A.4.2 Experiments

(E1): *[Profiler] [10 human-minutes + 10 compute-minutes]: extract a page-access trace from libjpeg.*
Preparation: Run `cd sgx-step/app/libjpeg`.
Execution: Run the profiler as follows:

```
$ sudo ../profiler/target/release/sgx_tracer \
    --so ./profiler-libjpeg.so \
    -e ./Enclave/encl.so \
    --output trace.vcd \
    --args img/birds.jpg 10000000 10000000
```

Collecting the full trace can take up to several hours. Interrupt the program after a few minutes.
Results: The partial trace in the `trace.vcd` file can be opened in a VCD viewer like GTKWave. This demonstrates the practicality of using the profiler to extract page-access traces and, thus, validates C1.

(E2): *[Attack on libjpeg] [10 human-minutes + 3 compute-hours]: reconstruct libjpeg image with page-fault attack.*
Preparation: Run `cd sgx-step/app/libjpeg/attack`.
Execution: Run the attack as follows:

```
$ cargo run --release -- -o reconstruct.bmp \
    -i ../img/Wapiti_from_Wagon_Trails.jpg \
    --color enclave \
    -e ../Enclave/encl.so
```

The attack can take several hours to complete, but progress is indicated by the progress bar.
Results: The resulting `reconstruct.bmp` file should contain a reconstruction of the original image, as in Figure 1, validating C2.

(E3): *[Prefetching simulation] [10 human-minutes + 30 compute-minutes]: use the instrumented binary to evaluate the effectiveness of TLBlur.*
Preparation:

```
$ export TLBLUR_LIB="$PWD/install/lib"
$ cd sgx-step/app/libjpeg
```

Execution: Run the profiler as follows:

```
$ sudo \
  ../profiler/target/release/sgx_tlblur_sim \
  --so $TLBLUR_LIB/libprof-libjpeg.so \
  -e $TLBLUR_LIB/s-encl-libjpeg-instrumented-
      ↪ relocs-bolt.so \
  --output trace_30.vcd \
  --args img/birds.jpg 10000000 10000000 \
  --pws-size 30 --irq-pat page-fault \
  --hw-tlb set-associative \
  --ways 8 --sets 1024
```

This will run the profiler with the instrumented libjpeg binary and a PWS size of 30, simulating the effect of TLBlur with a set-associative hardware TLB with 8 ways and 1024 sets. Collecting the full trace can take up to several hours. Interrupt the program after around 30 minutes have passed to collect sufficiently long traces.
Results: The partial trace in the `trace_30.vcd` file can be opened in a VCD viewer like GTKWave. Compared to the trace from E1, page-access patterns are "blurred", as in Figure 10, validating C3. Note that the binary instrumentation pass moves code to higher addresses, hence the code pages start at page 5120 instead.

(E4): *[Instrumentation benchmarks] [10 human-minutes + 2 compute-hours]: run performance benchmarks to measure instrumentation overhead.*
Preparation: Ensure that all benchmark enclave binaries are installed in `install/lib`. Each benchmark program and combination of optimizations shown in Table 3 has a corresponding `.so` file in this directory. Refer to the table in the README for the naming scheme of enclave files. Change to the `evaluation` directory.
Execution: To run the benchmarks, execute the `run_benchmarks.sh` script. This may take several hours to complete.
Results: The results can be found in the `out` directory. Run `./plot.py` to reproduce Figure 6. Set `FULL_PLOT = True` in `plot.py` to reproduce Table 3, validating C4.

(E5): *[Interrupt-resumption benchmark] [10 human-minutes + 5 compute-minutes]: benchmark the interrupt-resumption overhead.*
Preparation: Run the following to prepare the microbenchmark:

```
$ cd prefetch-benchmark
$ source /opt/intel/sgxsdk/environment
$ make clean all
```

Configure the enclave size in `Enclave/encl.config.xml`.
Execution: The `run.sh` script executes the microbenchmark for varying PWS sizes, and a fixed PAM size based

on the configured enclave size.

Results: The results can be found in the `out` directory. Repeat the experiment with different enclave sizes to obtain results for varying PAM sizes. Run `./plot.py` to reproduce Figure 7, showing linearly scaling overhead in both PWS and PAM size and, thus, validating C5.

A.5 Notes on Reusability

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Available Attestation: Towards a Reorg-Resilient Solution for Ethereum Proof-of-Stake

Mingfei Zhang
Shandong University
mingfei.zh@outlook.com

Rujia Li*
Tsinghua University
rujia@tsinghua.edu.cn

Xueqian Lu
Independent Researcher
xueqian.lu@bitheart.org

Sisi Duan*†
Tsinghua University
duansisi@tsinghua.edu.cn

A Artifact Appendix

A.1 Abstract

We present available attestation, a provable reorg-resilient solution for Ethereum Proof-of-Stake. In this work, we show that the majority of the known attacks on Ethereum PoS are some form of reorganization attacks. In practice, most of these attacks can be launched even if the network is synchronous (there exists a known upper bound for message transmission and processing). Different from existing studies that mitigate the attacks in an ad-hoc way, we take a systematic approach and provide an elegant yet efficient solution to reorganization attacks. We implement our modified protocol and five reorg attacks in our artifact and evaluate them using a local-built testnet of 16,384 clients. The results show that our protocol is resilient to five reorganization attacks and highly efficient. This artifact shows how to reproduce the results in Section 8 of our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All experiments are conducted on a local testnet (running on a single local machine). No experiments are conducted on the live Ethereum network. This repository does not uncover new vulnerabilities but instead analyzes known malicious reorganization attacks.

A.2.2 How to access

The artifact can be accessed by cloning our public Github project. All the scripts, container images, source codes, and sample output files can be accessed via the stable URL: https://zenodo.org/records/15205897.

A.2.3 Hardware dependencies

The experiments do not require any specialized hardware. Our test environment is a computer with a 4-core CPU, 16 GB of RAM, 100 GB of storage, and a 100 Mbps network connection.

A.2.4 Software dependencies

Our experiments run inside Docker containers. Make sure you install Docker following the official documentation, and verify that your Docker Engine version is at least 24.
We use Python for data processing and plotting. Make sure you have Python 3.10 (or a later version) installed. Then install the required packages via the following command::

```
pip3 install -r requirements.txt
```

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

After installing Docker and Python3, run the following steps:

1. Git clone the repository:
   ```
   git clone https://github.com/tsinghua-cel/available-attestation
   ```
2. Enter the cloned repository directory (denoted as `$HOME`):
   ```
   cd available_attestation/
   ```
3. Switch to the stable version:
   ```
   git checkout 94b76d76126481031e5aaa524cf19ced6414a182
   ```
4. Build the docker image:
   ```
   ./build.sh
   ```

* Corresponding author.
† Sisi is also with Zhongguancun Laboratory, Shandong Institute of Blockchains, BNRist, and State Key Laboratory of Cryptography and Digital Economy Security.

A.3.2 Basic Test

After building the docker image, run the basic test by the command:

```
./runtest.sh basic
```

The following outputs are expected:

```
[+] Running 19/19
 - Network basic_meta          Created      0.1s
 - Container execute5          Started      0.4s
 - Container execute3          Started      0.4s
 - Container execute1          Started      0.4s
 - Container ethmysql          Started      0.4s
 - Container execute2          Started      0.4s
 - Container execute4          Started      0.2s
 - Container beacon-2          Started      0.6s
 - Container attacker-1        Started      0.4s
 - Container beacon-3          Started      0.4s
 - Container beacon-1          Started      0.6s
 - Container beacon-4          Started      0.4s
 - Container beacon-5          Started      0.6s
 - Container validator-4       Started      0.6s
 - Container validator-3       Started      0.8s
 - Container strategy          Started      0.8s
 - Container validator-1       Started      0.8s
 - Container validator-2       Started      0.8s
 - Container validator-5       Started      0.8s
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The number of reorg blocks for the modified protocol is zero. This is validated by the experiments (E1), which reproduce the results described in Figure 13, Section 8 of the paper.*

(C2): *The throughput of the vanilla protocol and the modified protocol are almost the same. This is proven by the experiments (E2), which reproduce the results described in Figure 14, Section 8 of the paper.*

(C3): *The latency of the vanilla protocol and the modified protocol are almost the same. This is proven by the experiments (E3), which reproduce the results described in Figure 15, Section 8 of the paper.*

A.4.2 Experiments

(Overview) *We conduct three types of experiments: reorg resilience experiments (E1), throughput experiments (E2), and latency experiments (E3). Each experiment is performed for both the vanilla Ethereum PoS protocol and the modified Ethereum PoS protocol.*

(E1): *[25 compute-hour] The experiments involve five reorganization attacks (i.e., ex-ante reorg attack, sandwich reorg attack, unrealized justification reorg attack, justification withholding attack, and staircase attack). These experiments last for 12.5 hours on each protocol. The entire experimentation time thus lasts for 25 hours.*

How to: *run the experiment with the command under* `$HOME` *:*

```
./runtest.sh reorg
```

The experiments will run reorganization attacks for 9000 seconds on each protocol (approximately 25 hours in total). The output should be similar to that in the basic test. At the end of the experiment, the output will display:

```
test done and all data in
$HOME/results/reorgtest, report in
$HOME/reorgs.png
```

Results: *After completion, the result can be found in the* `$HOME` *directory. The number of reorg blocks for the modified protocol should align with the data shown in Figure 13 in the paper (i.e., the number of reorg blocks in the modified protocol is zero).*

Note: *Experiment E1 will run for 25 hours. If one does not want to wait for such a long period of time, run the following commands to conduct each attack separately. Each attack lasts for about five hours.*

Run the modified exante reorg attack:

```
./runtest.sh 1
```

Run the sandwich reorg attack:

```
./runtest.sh 2
```

Run the unrealized justification reorg attack:

```
./runtest.sh 3
```

Run the justification withholding reorg attack:

```
./runtest.sh 4
```

Run the staircase attack:

```
./runtest.sh 5
```

(E2): *[1.3 compute-hour] The experiments are used for assessing the throughput of both the vanilla protocol and the modified protocol.*

How to: *run the experiment with the command under* `$HOME` *:*

```
./runtest.sh tps
```

An experiment lasts for 40 minutes for each protocol. The entire experiment lasts for 1.3 hours in total. The output should be similar to that in the basic test. At the end of the experiment, the output will display:

```
test done and all data in $HOME/results/tps,
report in $HOME/tps.png.
```

Results: *After completion, the results can be found in the* `$HOME` *directory. The throughput of the modified protocol should align with the data shown in Figure 14 in the paper (i.e., the throughput of the vanilla protocol and the modified protocol are almost the same).*

(E3): *[20 compute-minutes]: The experiments test the latency of both the vanilla protocol and the modified protocol.*

How to: *run the experiment with the command under* `$HOME` *:*

```
./runtest.sh latency
```

The experiments evaluate the computation time across

both the vanilla protocol and the modified protocol. The output should be similar to that in the basic test. At the end of the experiment, the output will display:

```
test done and all data in
$HOME/results/blockcost, report as bellow.
```

Results: *After completion, the results can be found in the* `$HOME` *directory. The latency of the modified protocol should align with the data shown in Figure 15 in the paper (i.e., the latency of the vanilla protocol and the modified protocol are almost the same).*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Voting-Bloc Entropy: A New Metric for DAO Decentralization

Andrés Fábrega[1], Amy Zhao[3], Jay Yu[5], James Austgen[2], Sarah Allen[3,4], Kushal Babel[2,3], Mahimna Kelkar[2], Ari Juels[2,3]

[1] Cornell University [2] Cornell Tech [3] IC3 [4] Flashbots [5] Stanford University

A Artifact Appendix

This appendix describes how to use the artifacts included in *Voting-Bloc Entropy: A New Metric for DAO Decentralization*. The core contribution of our paper is Voting-Bloc Entropy (VBE), which is a new framework for computing the degree of decentralization in DAOs. At a high-level, VBE consists of two main steps:

1. Computing a *clustering function* over the utility functions of the members of the DAO. This step results in groups—or *blocs*—of DAO members with aligned incentives and interests, as determined by the similarity of their utility functions.

2. Computing an *entropy metric* over the distribution of voting power (i.e., DAO tokens) across the resulting blocs.

Since VBE is a framework, it can be instantiated with any clustering function and entropy metric.

Note that utility functions are so-called latent variables, which cannot be directly measured. So, to compute VBE in practice, we need to use some on-chain, observable data to estimate the utility functions of the members of the DAO, such as voting history. This leads to the "observable" VBE (oVBE) framework, which computes the clustering step using some relevant dataset that represents utility functions.

A.1 Abstract

The main component of our artifact is the *oVBE toolkit*, which is a library for computing various instantiations of oVBE, supporting some of the most popular clustering algorithms (e.g., K-means, hierarchical clustering, DBSCAN, etc) and entropy functions (e.g., min-entropy, Shannon entropy, etc). Using our library, users can compute the oVBE of a DAO, and understand its degree of decentralization or perform VBE-based governance experiments.

At a high-level, to use our toolkit, users simply input a data set of observable data, and select appropriate parameters for clustering and entropy. While in principle our toolkit is compatible with any dataset, most users will probably be interested in voting history as the input data. As such, our artifact also includes scripts for easily scraping the voting history of DAOs from the blockchain, as well as for preparing it in a format that is compatible with the rest of the VBE toolkit.

In the rest of this appendix, we describe the general use of our artifact, as well as how to use it specifically to reproduce the experiments of our paper.

A.2 Description & Requirements

Our artifact relies on fairly standard software dependencies, which can run on typical consumer laptops. We summarize the requirements in this section.

A.2.1 Security, privacy, and ethical concerns

There are no security or data privacy risks associated with evaluating our artifact. In particular, we have provided API keys for the reviewers, so that they do not need to provide personal information to register with the services that our toolkit leverages.

A.2.2 How to access

The artifact stable repository can be found at `https://zenodo.org/records/14675832?token=eyJhbGciOiJIUzUxMiJ9.eyJpZCI6ImY3ODk0ZGF%5B%E2%80%A6%5DIQ1CNdjCqd1D4NC7ldSWtrYyVog-sJusoe9Ma2tfu82NEJv2sJo4UO8bpOBcpEw`.

Note that, as described further below, API keys for Tally and Snapshot are necessary to use some parts of our artifact.

A.2.3 Hardware dependencies

Our artifact does not require any special hardware features.

A.2.4 Software dependencies

Our artifact requires Python 3.11 (at least), and installation of the Python libraries specified in file `https://github.com/DAO-Decentralization/VBE/blob/main/VBE-data/requirements.txt`.

We tested our artifact on MacOS and Ubuntu, but believe it should run on any platform that can install the aforementioned Python libraries. If using Ubuntu, additional dependencies need to be installed running "sudo apt install python3-venv libpq-dev", as described in https://github.com/DAO-Decentralization/VBE/tree/main/VBE-data/README.md.

A.2.5 Benchmarks

To reproduce our experiments, we have made available our data in the form of a read-only database, the details of which can be found at https://github.com/DAO-Decentralization/VBE/blob/main/VBE-data/.env.example.

A.3 Set-up

A.3.1 Installation

To install our artifact, follow the instructions in https://github.com/DAO-Decentralization/VBE/blob/main/VBE-data/README.md followed by the instructions in https://github.com/DAO-Decentralization/VBE/blob/main/VBE-library/README.md.

A.3.2 Basic Test

Using the example included in https://github.com/DAO-Decentralization/VBE/blob/main/VBE-data/data_setup/dao_input.csv, run "python tally_api.py" or "python snapshot_api.py". This should successfully output files into VBE-data/data_output/ folder if selecting the option to save as CSV. This tests the basic functionality of our data processing scripts

Using the resulting data output, one can then test the oVBE toolking by following the instrictions in https://github.com/DAO-Decentralization/VBE/tree/main/VBE-library to run either vbe_parameters.py or run_vbe.py for a single or full instance of calculations.

A.4 Evaluation workflow

This section describes how to use and evaluate our artifact. The high-level workflow of our artifact proceeds as follows:

- First, use the scripts in the VBE-data directory to acquire governance data for DAOs of interest, which will be the input to the VBE library. This data will be scraped using Tally and Snapshot, for which API keys are required. The scrapped data will be saved either in a database endpoint specified by the user, or output as a CSV file. Note that this step is optional, as users that already have governance data can directly use the oVBE toolkit.
- Use the scripts in VBE-library to compute oVBE based on some input data (which can potentially be the output of the prior step). In particular, to reproduce the results of our paper, we have made available a read-only database with the data used by our results.

A.4.1 Major Claims

The results in our paper that rely on our artifact are the following:

1. Average minimum entropy VBE values for DAOs listed as follows in Figure 1: Arbitrum (0.9254), Optimism (0.9929), Gitcoin (0.8536), Nouns DAO (0.8806), and Uniswap (0.7435). This is proven in Experiment 4 (E4) where VBE is calculated using default parameters across all organizations and proposals in windows.

2. Voter participation, Nakamoto Coefficient and Gini Index from Figure 1 are from Experiment 3 (E3) to aggregate proposal and DAO-level information.

A.4.2 Experiments

Experiment #1: pull Snapshot data. This step is used to pull information from the Snapshot API for all DAOs listed in the input file.

1. Preparation: Steps for dependencies and configuration can be found in https://github.com/DAO-Decentralization/VBE/blob/main/VBE-data/README.md. Make sure to have a Snapshot API key in the .env file.

2. Execution: run "python snapshot_api.py", specify whether to save locally or to a database, and view outcomes in the database tables or in data_output/ folder.

3. Results: Depending on the data_input.csv specified, expect to see the corresponding data for DAOs, proposals, and voter information.

Experiment #2: pull Tally data. This step is used to pull information from the Tally API for all DAOs listed in the input file.

1. Preparation: Steps for dependencies and configuration can be found in https://github.com/DAO-Decentralization/VBE/blob/main/VBE-data/README.md. Make sure to have a Tally API key in the .env file.

2. Execution: run "python tally_api.py", specify whether to save locally or to a database, and view outcomes in the database tables or in data_output/ folder.

3. Results: Depending on the data_input.csv specified, expect to see the corresponding data for DAOs, Proposals, and Voter information.

Experiment #3: aggregate data analytics. This step is used to pull information from database or CSVs and turn them into aggregate level statistics

1. Preparation: As a prerequisite, have data pulled from Snapshot or Tally as per the two experiments above.
2. Execution: run "python data_analytics.py", specify whether to save locally or to a database, and view aggregated data in the database tables or in data_output/ folder.
3. Results: The aggregate data includes information like the voter participation rate, number of proposals, percentiles for voting power, and voter choices.

Experiment #4: compute oVBE. This is the main experiment in our artifact, which computes oVBE on an input database or CSV file.

1. Preparation: Follow steps outlined in the VBE-library README file. If using a database connection, make sure it is properly configured in the .env file.
2. Execution: run python run_vbe.py, specify whether to save locally or to a database, and view outcomes in the database tables or in data_output/ folder.
3. Results: Results are output to the data_output/ folder, where the main information for VBE is saved to dao_vbe.csv or the dao_vbe table in the database.

The database endpoint containing the data used for our experiments (which can be specified in the .env file) can be found in https://github.com/DAO-Decentralization/VBE/blob/main/VBE-data/.env.example.

A.5 Notes on Reusability

Edit the run_vbe.py to have a different model, different parameters. Can edit load_data.py to accommodate different input data, and featurize information for clustering and VBE calculation. Add more scripts in the future (beyond Snapshot and Tally) to bring in other sources for governance data. Use vbe_parameters.py to run a wide range of clustering and entropy calculations - this (and utils.py) can be configured in the future to include more options not listed. Rds_readonly.py can also be used as a starting point to read and manipulate data in the database. If needed, custom queries can be written to extract specific information.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: **Let's Move2EVM**

Lorenzo Benetollo[*‡§¶], Andreas Lackner[*†], Matteo Maffei[†¶], Markus Scherer[†¶]
[†]TU Wien
[‡]Ca' Foscari University of Venice
[§]University of Camerino
[¶]Christian Doppler Laboratory Blockchain Technologies for the Internet of Things

A Artifact Appendix

A.1 Abstract

The Move programming language, designed with strong safety guarantees such as linear resource semantics and borrow-checking, has emerged as a secure and reliable choice for writing smart contracts. However, these guarantees depend on the assumption that all interacting contracts are well-formed—a condition naturally met in Move's native execution environment but not in heterogeneous or untrusted platforms like the Ethereum Virtual Machine (EVM). This work addresses the challenge of preserving Move's security guarantees when compiling to EVM.

We provide multiple artifacts to support the claims made in the paper and to facilitate data reproduction. In particular, the following artifacts are included:

- **Move-to-EVM-Compiler:** The source code of our adaptations of the original Move-to-EVM Compiler.
- **Datasets:** The Rosetta and ERC-20 Dataset used in the evaluation. Due to licensing issues, we cannot provide data from the Aptos benchmark. However, we provide scripts to build the set from publicly available sources.
- **Supplementary artifacts:** Scripts and tools to simplify running our experiments.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our experiments do not perform any destructive operations and do not collect any private or sensitive data. To reproduce the Aptos benchmark used in the paper, we rely solely on publicly available information. Data scraping is conducted conservatively to avoid overloading the target system.

[*]Shared first authorship.

A.2.2 How to access

Our artifacts are publicly available at `https://doi.org/10.5281/zenodo.15591737`. The artifact consists of the following files:

- `aptos_dataset.zip`: A zip file containing parts of the aptos dataset and tools to generate the full dataset.
- `docker.zip`: A zip file containing additional files needed to create the Docker image.
- `Dockerfile`: The Dockerfile to create an image containing our compiler.
- `evaluation.zip`: A zip file containing tools and data to reproduce our results.
- `gas_prices.zip`: A zip file containing gas prices that we refer to in the paper.
- `lets-move-to-evm.zip`: A zip file containing the source code of our compiler.
- `paper_extended.pdf`: The extended version of the paper.
- `playground.zip`: A playground to compile Move packages in general.
- `README.md`: Additional documentation on how to run the compiler in general.

In the following descriptions, we assume that you have downloaded all files of the artifact and unzipped the (top-level) zip files mentioned above.

A.2.3 Hardware dependencies

System Requirements Artifacts can be run on both x86 and ARM architectures. We recommend 8 GB of RAM and at least 20 GB of available disk space.

A.2.4 Software dependencies

Operating System Experiments were conducted on Ubuntu 24.04. Other systems might work, but may require additional adjustments.

Docker The compiler runs inside a Docker container. All experiments were tested with Docker version 27.5.1, although newer versions will also work. To download the latest version, we refer to the official documentation: https://www.docker.com/get-started/.

Nix We use the Nix package manager for parts of our evaluation. We refer to https://nixos.org/download/ for installation instructions. To reproduce our environment, we suggest version 2.18.1 or higher.

Avoiding Nix If you have the Zstandard CLI installed and Python 3 with the `requests` package, you might still be able to run our experiments. However, please note that we have not tested this setup.

A.2.5 Benchmarks

The benchmark datasets for Rosetta and ERC-20 are included in the artifact. To retrieve the Aptos dataset, we refer to **A.3.1** for detailed instructions.

A.3 Set-up

A.3.1 Installation

Download files and install dependencies First, download the artifact from https://doi.org/10.5281/zenodo.15591737 in any directory and unzip the zip files as described before. From now on, we assume that your working directory looks as follows (assuming you have deleted unzipped zip files):

```
aptos_benchmark
├── docker
├── evaluation
├── gas_prices
├── lets-move-to-evm
├── playground
├── Dockerfile
├── paper_extended.pdf
└── README.md
```

Preparing benchmarks For the Aptos benchmark, due to licensing issues, we cannot directly provide the source files. Anyhow, we provide scripts to download and patch the files:

```
cd aptos_benchmark
nix-shell
./build_eval.bash
```

Note that two modules (`offer` and `universal_config`) will be skipped. This is as expected and relates to compiler limitations described in the paper. The results of this script are copied to the `evaluation` folder. After that, you can leave the nix shell again by pressing `CTRL+d`.

Building the Docker image The Docker image contains the compiler and all its dependencies. Building the Docker container can last up to 30 minutes. Note that we expect you to execute the next line in the root folder of your downloaded artifact (where the `Dockerfile` is located)

```
docker build -t mv2evm .
```

This should build a new Docker image containing the compiler. You can verify whether everything was successful by running `docker image ls`. The list of images should contain `mv2evm`.

A.3.2 Basic Test

After successfully running the previous steps, you can execute the following command to see if everything worked correctly:

```
./evaluation/attach.bash <COMPILER> <DOCKERIMG>
```

For `<DOCKERIMG>` use `mv2evm` as created before. For `<COMPILER>`, test it for each of the values `IRM`, `ORIGINAL`, `SOL`. Executing the script `attach` should open a shell (inside the created Docker container). Note that it might take up to one minute until the Docker container is started.

Inside the new shell you should be able to execute the command `move --help` which should return a list of available command-line-options. **You can leave the Docker container now by typing** `CTRL+d`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1) From Section 2.3: The original compiler is vulnerable to the bugs referred to as C1, C2, and C4 in the paper (names unfortunately collide with the Major Claims in this section). This is proven by (E1).

(C2) From Section 4.2 §ERC-20: When using Move in an *unidiomatic way* (as proposed by the original compiler), gas costs nearly double with respect to the ERC-20 version of Solidity (C2.1). Using Move as intended leads to an overhead of at most 42% (C2.2) when using the (insecure) original compiler. Compiled with our compiler, we get an overhead of at most 50% (C2.3), where only 15% are caused by our changes in the compiler. This can be proven by (E2).

(C3) From Section 4.2 §Rosetta: For the Rosetta benchmark, all claimed contracts (see Figure 7) can be compiled by our compiler (C3.0). On average, the overhead

compared to the original compiler is less than 49,000 (C3.1). Compared to Solidity, it is less than 98,000. This is proven by (E3).

(C4): From Section 4.2 §Aptos: For the Aptos benchmark, all claimed contracts (see Figure 8) can be compiled by our compiler (C4.0). The overhead compared to the original compiler is on average around 12,000 of gas (C4.1). This is proven by (E4).

A.4.2 Experiments

NOTE: In this section, the working directory will be the `evaluation` directory!

(E0): Execute all tests [30 human-minutes + 10 compute-minutes]: This experiment is the basis for most of the other experiments. It must be repeated for each of the three compilers. We use `<COMPILER>` to indicate the values `IRM`, `SOL`, and `ORIGINAL`.
Preparation: Attach to the Docker container by typing `./attach.bash <COMPILER> mv2evm`.
Execution: Execute the script `./evaluate.bash` that should be placed in the current working directory of the just-opened shell inside the Docker container.
Results: The script executes test cases. There must not be any failing test cases (failing test cases would be noted at the end of the output). Also, typing `ls ./results` must yield a non-empty list of `csv` files. Warnings are expected and can be ignored.

(E1): Vulnerability Check [<10 human-minutes + <10 compute-minutes]:
Preparation: Attach to the Docker container by typing `./attach.bash ORIGINAL mv2evm`
Execution: Execute the script `./vul_check.bash` that should be placed in the current working directory of the just-opened shell inside the Docker container.
Results: The script should execute three tests. All of them must be passed (indicated by `3 passing`). Compiler warnings are expected and can be ignored.

(E2): ERC-20 [30 human-minutes + <10 compute-minutes]:
Preparation: First execute (E0). Then leave the Docker container (`ctrl+d`). You should now be in folder `evaluation`.
Execution: Execute the script `./e2.bash`.
Results: We expect the following output: 88% for (C2.1); 28% for (C2.2); 36% for (C2.3); and 20% for (C2.4). We note that numbers are slightly **better** than those presented in the paper since the Solidity version performs worse in our test environment. Other numbers are unchanged (compare output with Figure 6 of the paper).

(E3): Rosetta [30 human-minutes + <10 compute-minutes]:
Preparation: First execute (E0) if not done previously. Then leave the Docker container (`ctrl+d`). You should now be in folder `evaluation`.
Execution: Execute the script `./e3.bash`.
Results: We expect the following output: 35,482 for (C3.1); and 89,942 for (C3.2). Regarding C3.0, the script should output results for all contracts listed in Figure 7. We note that the numbers for IRM slightly differ from the paper due to fixes. However, in case of increasing costs, the increase should be smaller than 3% (see last column printed by the script).

(E4): Aptos [30 human-minutes + <10 compute-minutes]:
Preparation: First execute (E0) if not done previously. Then leave the Docker container (`ctrl+d`). You should now be in folder `evaluation`.
Execution: Execute the script `./e4.bash`.
Results: We expect a value of 12,174 for C4.1. Regarding C4.0, the script should output results for all contracts listed in Figure 8. Numbers should match those of the paper up to the nearest thousand (as represented in the paper).

A.5 Notes on Reusability

For the reproducibility of the data mentioned in the paper, we only need a small part of the Aptos dataset. However, in the `README.md` of the artifact, we provide detailed instructions on how to retrieve the whole dataset.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Surviving in Dark Forest: Towards Evading the Attacks from Front-Running Bots in Application Layer

Zuchao MA

A Artifact Appendix

A.1 Abstract

Real-world strategies for evading front-running remain under-explored in their taxonomy and distribution due to their covert nature. Understanding these evasion tactics is vital for assessing the resilience of the current blockchain application layer and identifying areas for potential enhancement, thereby strengthening the ecosystem. In this work, we take the first step to demystify evading strategies in Ethereum and BNB Smart Chain. We propose EVScope, a novel framework combining binary analysis and machine learning to detect known and unknown evading strategies. The artifact includes the source code of EVScope, the transactions that adopt evading strategies, and the guidance to recreate results and run tools. The goals to be reproduced in the artifact evaluation include:

• Our findings uncover 32 refined strategies involving access control, profit control, execution split, and code obfuscation (Table 3 in RQ1).

• The F1-measure of EVScope reaches 0.97 on average (Table 4 in RQ4).

• We develop a straw-man approach (SA) as the baseline to conduct the comparison with EVScope. The F1-measure of SA reaches 0.30 on average (Table 4 in RQ4).

• The clustering accuracy in iterations (Figure 14 in RQ4).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The original artifact can be accessed in https://zenodo.org/records/14735789. In the artifact availability evaluation, reviewers provide some useful comments to optimize our `readme` file, and I update the `readme` file in the artifact to create version V2 (https://zenodo.org/records/14778705).

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Please use Ubuntu OS and docker for executing the artifact. In our prepared docker image `evasion.tar`, we already set all software dependencies.

A.2.5 Benchmarks

The dataset of evading strategies is in `src_rqs.zip/RQs/dataset`. We also prepare the dataset in the docker image `evasion.tar`, which means that reviewers have no need to collect dataset.

A.3 Set-up

A.3.1 Installation

Installing docker in Ubuntu OS, for example docker 27.3.1.

A.3.2 Basic Test

Use the command to check whether docker has been installed.
`docker -version`
The expected output should be like
`Docker version XXX (e.g., 27.3.1), build YYY (e.g., ce12230)`
Import docker image by the command
`sudo docker load < evasion.tar`
Then you can see the image id of the image by the command
`sudo docker images`
Assume the image id is [image-id]. Start the container by the command
`sudo docker run -itd -name evasion [image-id]`
We can see the container id of the container by the command
`sudo docker ps`
Assume the container id is [container-id]. Go into the container by command
`sudo docker attach [container-id]`
After going into the container, please start mongodb database for querying results by the command

```
mongod -f /etc/mongod.conf &
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our findings uncover 32 refined strategies involving access control, profit control, execution split, and code obfuscation. This is proven by the experiment described in in Section 5.2 (RQ1) whose results are illustrated in Table 3.

(C2): The F1-measure of EVScope reaches 0.97 on average. This is proven by the experiment described in Section 5.5 (RQ4) whose results are illustrated in Table 4.

(C3): We develop a straw-man approach (SA) as the baseline to conduct the comparison with EVScope. The F1-measure of SA reaches 0.30 on average. This is proven by the experiment described in Section 5.5 (RQ4) whose results are illustrated in Table 4.

(C4): The clustering accuracy increases in iterations. This is proven by the experiment described in Section 5.5 (RQ4) whose results are illustrated in Figure 14. Considering terminal interface cannot illustrate the figure, the artifact prints the points of the curves of Figure 14 in terminal for review.

A.4.2 Experiments

(E1): [1 human-minute + 5 compute-minutes + 60GB disk]: run two scripts to generate the content of Table 3 to two csv files. Then use `cat` command to show the content of Table 3 on the terminal.
Preparation: In the container, go into the folder `/home/ubuntu/RQs` by the command
`cd /home/ubuntu/RQs`
Execution: Execute the script `gen_table.py` to generate a `table.csv` file that contains line 1 to 24, line 26 to 30 of Table 3, by the command
`python3 gen_table.py`
Execute the script `gen_tab_lock.py` to generate a `table_lock.csv` file that contains line 25, line 31 to 32 of Table 3, by the command
`python3 gen_tab_lock.py`
Results: Print the content of `table.csv` to terminal by the command
`cat table.csv`
Then compare the content of terminal with Table 3 (line 1 to 24, line 26 to 30) of the paper to check the result.
Print the content of `table_lock.csv` to terminal by the command
`cat table_lock.csv`
Then compare the content of terminal with Table 3 (line 25, line 31 to 32) of the paper to check the result.

(E2): [1 human-minute + 1 compute-minute]: run a script to generate the content of Table 4 (EVScope row).
Preparation: In the container, go into the folder `/home/ubuntu/RQs` by the command
`cd /home/ubuntu/RQs`
Execution: Execute the script `precision.py` to generate the the result of EVScope row in Table 4, by the command
`python3 precision.py`
Results: Compare the content of terminal with Table 4 (EVScope row) of the paper to check the result.

(E3): [1 human-minute + 1 compute-minute]: run a script to generate the content of Table 4 (Straw-man row).
Preparation: Go into the folder `/home/ubuntu/RQs` by the command
`cd /home/ubuntu/RQs`
Execution: Execute the script `strawman.py` to generate the the result of Straw-man row in Table 4, by the command
`python3 strawman.py`
Results: Compare the content of terminal with Table 4 (Straw-man row) of the paper to check the result.

(E4): [1 human-minute + 1 compute-minute]: run a script to generate the content of Figure 14.
Preparation: Go into the folder `/home/ubuntu/RQs` by the command
`cd /home/ubuntu/RQs`
Execution: Execute the script `cluster_accuracy.py` to generate the the data points in Figure 14, by the command
`python3 cluster_accuracy.py`
Results: Compare the data points of terminal with Figure 14 of the paper to check the result.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Unlocking the Power of Differentially Private Zeroth-order Optimization for Fine-tuning LLMs

Ergute Bao[*], *Yangfan Jiang*[†], *Fei Wei*[*], *Xiaokui Xiao*[†], *Zitao Li*[*], *Yaliang Li*[*], *Bolin Ding*[*]
[*]*Alibaba Group,* [†]*National University of Singapore*

A Artifact Appendix

A.1 Abstract

The main idea of our proposed method, DP-AggZO, is to aggregate **multiple zeroth-order estimates** for the exact gradients, computed over independent perturbation vectors (random Gaussian vectors), before enforcing differential privacy (i.e., artificial clipping, taking the average, and then injecting random DP noises). Compared with the vanilla DPZO (or DPZero), which is effectively a degenerated version of DP-AggZO with only **one** zeroth-order estimate, our DP-AggZO achieves much better utility under the same privacy constraints. Our DP-AggZO also outperforms the state-of-the-art DP-AdamW in some cases. This artifact is used for validating the above claim.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Security, privacy, and ethical concerns are not applicable to this artifact as the models and datasets are publicly available.

A.2.2 How to access

Access via Zenodo in https://zenodo.org/records/15594622.

A.2.3 Hardware dependencies

A workstation or cloud computing node with a GPU (preferrably with GPU memory larger than 20 GB), e.g., RTX 4090 GPU 24GB, or above (larger GPU memory is needed if run larger models, e.g., OPT 6.7B).

A.2.4 Software dependencies

Linux system Ubuntu 22.04.4, installed with python 3.9.18, with torch==2.4.0+cu121, transformers==4.28.1, and opacus==1.4.0. More on enviroments can be found in the file named "environments.yml" provided.

A.2.5 Benchmarks

We evaluate on RoBERTa (355M) [1], which is a pretrained model on English language using a masked language modeling (MLM) objective, and OPT-1.3B and OPT-6.7B [2], which are parts of the a suite of decoder-only pre-trained transformers ranging from 125M to 175B parameters.

In the provided script, there are on six datasets for different classification tasks, including SST-2 and SST-5 [3] for sentiment analysis (i.e., determine if the given text is positive/negative), SNLI [4], MNLI [5], and RTE [6] for natural language inference (i.e., determine if the given premise and hypothesis are in the relationship of entailment/neutral/contradiction), and TREC [7] for topic assignment (i.e., determine which topic the given question falls under). For each dataset, we generate 512 samples for each class for training.

For larger models OPT-1.3B and OPT-6.7B that require more resources to fine-tune, we focus on one classification task SST-2 and one generation task SQuAD [8] (the fine-tuned model answers to a given question containing relevant contexts). For each dataset, we use 1000 samples for training.

A.3 Set-up

A.3.1 Installation

Please Use the file named "environments.yml" provided to install the environment using pip or conda. For testing RoBERTa (355M), go to folder roberta and install the dataset as specified in the readme file provided.

A.3.2 Basic Test

After installation of the environment and datasets, to test the functionality, run the following script in bash.

[1] https://huggingface.co/FacebookAI/roberta-large
[2] https://huggingface.co/facebook/opt-1.3b
[3] https://aclanthology.org/D13-1170/
[4] https://aclanthology.org/D15-1075/
[5] https://aclanthology.org/N18-1101/
[6] https://dl.acm.org/doi/10.1007/11736790_9
[7] https://aclanthology.org/L00-1018/
[8] https://aclanthology.org/D16-1264/

```
CUDA_VISIBLE_DEVICES=0 DPZERO_PRIVACY_EPS=2
    DP_SAMPLE_RATE=0.0208 STEP=500 \
SEED=42 NUM_DIRECTION=64
    RANDOM_DIRECTION_SEED=100 LR=6e-4 \
DPZERO_THRESHOLD=1 TASK="MNLI" bash examples
    /dpaggzo.sh
```

This script takes around 80 minutes to run on a RTX 4090 GPU and gives a test accuracy around 72%, which is much better than the utility of the original DPZO/DPZero under the same privacy constraint (around 65%, or refer to Table 2 on Page 9 of their original paper [9]).

Results on other datasets can be obtained by changing the "TASK" parameter. We also provided more examples in the "readme" file.

A.4 Evaluation workflow

A.4.1 Major Claims

We make two major claims.

(C1): *Our DP-AggZO can outperform the vanilla DPZero/DPZO in terms of test accuracy under the same privacy constraints.*

(C2): *Our DP-AggZO can sometimes outperform DP-AdamW in terms of test accuracy under the same privacy constraints.*

A.4.2 Experiments

C1-1: Reproducing DP-AggZO results for MNLI with privacy level $\varepsilon = 2$. You can run the DP-AggZO experiments for the MNLI task directly using the following commands:

```
CUDA_VISIBLE_DEVICES=0 DPZERO_PRIVACY_EPS=2
    DP_SAMPLE_RATE=0.0416 STEP=1000 \
SEED=42 NUM_DIRECTION=64
    RANDOM_DIRECTION_SEED=100 LR=8e-5 \
DPZERO_THRESHOLD=5 TASK="MNLI" bash examples
    /dpaggzo.sh
```

or

```
CUDA_VISIBLE_DEVICES=0 DPZERO_PRIVACY_EPS=2
    DP_SAMPLE_RATE=0.0416 STEP=1000 \
SEED=42 NUM_DIRECTION=64
    RANDOM_DIRECTION_SEED=100 LR=5e-4 \
DPZERO_THRESHOLD=1 TASK="MNLI" bash examples
    /dpaggzo.sh
```

Expected outcome:

- Compute time: 5 hours on RTX 4090, 14 hours on RTX A5000.
- Test accuracy is ∼74% on RTX A5000 or H20 GPU; ∼71% on RTX 4090 GPU.
- Significantly better than vanilla DPZO/DPZero under the same privacy level (see below).

C1-2: Reproducing DP-AggZO results with $K = 1$ (equivalent to DPZero/DPZO baseline) with privacy level $\varepsilon = 2$.

```
CUDA_VISIBLE_DEVICES=0 DPZERO_PRIVACY_EPS=2
    DP_SAMPLE_RATE=0.0416 STEP=5000 \
SEED=42 NUM_DIRECTION=1
    RANDOM_DIRECTION_SEED=100 LR=2e-6 \
DPZERO_THRESHOLD=200 TASK="MNLI" bash
    examples/dpaggzo.sh
```

Expected outcome:

- Compute time: 40 minutes on RTX 4090
- Test accuracy: 65% on RTX 4090 GPU, or can refer to the original paper: arxiv.org/pdf/2310.09639

Combining the results of **C1-1** and **C1-2**, we can verify that DP-AggZO outperforms the vanilla DPZO/DPZero under the same privacy constraints.

C2-1: Reproducing DP-AggZO results for MNLI with privacy level $\varepsilon = 0.5$. You can run the DP-AggZO experiments for the MNLI task directly using the following command:

```
CUDA_VISIBLE_DEVICES=0 DPZERO_PRIVACY_EPS
    =0.5 DP_SAMPLE_RATE=0.0416 STEP=500 \
SEED=42 NUM_DIRECTION=64
    RANDOM_DIRECTION_SEED=100 LR=2e-4 \
DPZERO_THRESHOLD=1 TASK="MNLI" bash examples
    /dpaggzo.sh
```

Expected outcome:

- Compute time: 3 hours on RTX 4090, 7 hours on RTX A5000.
- Accuracy: ∼63.5% on H20 and A5000 GPUs
- Better than DP-AdamW under the same privacy level (∼ 62%) (see below).

C2-2: Reproducing the result on DP-AdamW using the following fommand:

```
CUDA_VISIBLE_DEVICES=0 DP_SAMPLE_RATE=0.0416
     STEP=1000 SEED=42 LR=1e-4 \
DPSGD_THRESHOLD=10 DPSGD_PRIVACY_EPS=0.5
    DPSGD_PRIVACY_DELTA=1e-5 \
TASK="MNLI" bash examples/dpsgd.sh
```

The results from **C2-1** and **C2-2** demonstrate that **DP-AggZO can outperform DP-AdamW** the same privacy constraints, but the improvement is not as significant as that for DPZO/DPZero. We refer to the "readme" file provided for more examples.

To use this artifact beyond the models presented in

[9] https://arxiv.org/pdf/2310.09639

A.5 Notes on Reusability

To use this artifact beyond the models presented in this paper, we would recommend refactoring the code based on the latest implementation and algorithms of the model of interest. Some functions/libraries may become obsolete in the future while the general algorithmic idea of using multiple independent zeroth-order estimates to reduce clipping error could still apply.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Evaluating LLM-based Personal Information Extraction and Countermeasures

Yupei Liu, Yuqi Jia, Jinyuan Jia, and Neil Zhenqiang Gong

A Artifact Appendix

This is the artifact appendix for our USENIX Security'25 paper. This appendix is meant to describe a roadmap for the readers to understand what our artifact is and how to properly use our artifact, including any necessary environment.

A.1 Abstract

Our artifact contains three components: 1) the code for main experiments, 2) the code for baseline comparison, and 3) the Conda environment we used for evaluation.

A.2 Description & Requirements

In this section, we describe details of how to access our artifact, the hardware/software dependencies of our artifact, and any benchmarks in our artifact.

A.2.1 Security, privacy, and ethical concerns

As discussed in the Phase-1 AE, README in the "official_code" directory in the Zenodo URL, and the appendix of our paper, we will not share the other 3 datasets (i.e., Celebrity, Physician, and Professor) by default. Instead, we only share them per request to verified researchers. This is due to the dual nature of the release of these real-world datasets. In particular, they can benefit both defenders and attackers. Therefore, in this artifact, we don't provide the data or any configuration/code related to these 3 datasets.

A.2.2 How to access

The URL to access our main artifact is: https://zenodo.org/records/14737200. This repository contains three files/directories at root: 1) README, 2) official_code which is the code/data for our main experiments, and 3) baseline_code which contains our implementation to the baseline methods compared in our paper.

In addition, the URL to the Conda environment we used for our experiments is available at: https://zenodo.org/records/14868933.

A.2.3 Hardware dependencies

We conducted most of experiments on a single Quadro RTX 6000 GPU. Less powerful GPUs should also be able to run some experiments using our artifact on smaller LLMs. We note that for Flan-UL2 experiments, we use a single RTX Titan GPU. Regarding open-source models studied in our paper and released in our artifact, they can be executed even with only fairly poor computational resources like CPUs, because these models' results are obtained by querying remote models using their APIs. Moreover, we want to note that we used x86_64 platform for our experiments. To run the artifact on other archs like aarch64, the libraries may differ.

A.2.4 Software dependencies

For experiments on open-source LLMs, third-party softwares or APIs are required. For Google's Gemini pro, an account to Google's cloud platform is required. For OpenAI's GPT models, we use Microsoft Azure APIs for deployment and querying. For PaLM2 models, we leveraged Google's cloud APIs when we did the experiments. However, we note that PaLM2 models are no longer available on Google's platforms now. Instead, users may leverage a third-party deployment such as VertexAI or ClarifAI.

The URL to the Conda environment we used for our evaluation is at: https://zenodo.org/records/14868933. We use glnxa64 and our CUDA version is 12.1.

A.2.5 Benchmarks

Our paper does not produce any new models in our artifact. However, we indeed released the Synthetic dataset used in our evaluation. It is described in the README of the "official_code" directory. In addition, we provide the code to download the "Court" dataset which is from a previous work.

Again, we want to clarify that for ethical concerns, we don't release the other three datasets (i.e., Celebrity, Physician, and Professor) by default, because these datasets contain real-world persons' information. They are only shared per request with verified researchers. Future researchers who are interesed in obtaining access to these datasets are expected to contact the authors using institutional email addresses and explain the main purpose of using these datasets. This way of sharing these three real-world datasets, as discussed previously with

reviewers, is to guarantee that they are not misused by any potential malicious parties.

A.3 Set-up

The setup should involve preparing the right hardware such as GPUs for the artifact. Please refer to A.2.3 for more details.

A.3.1 Installation

Users can first download the "PIE_environment.yml" from the URL mentioned above. Then, users can use Conda command such as "conda env create –name envname –file=PIE_environment.yml" to create a virtual environment from this file. Next, users can download the code and data from https://zenodo.org/records/14737200 and extract the .zip files.

A.3.2 Basic Test

Users can step into "official_code" directory and run "python3 run.py" for a basic test. Note that this will run the Vicuna-13b-v1.3 model on the Synthetic dataset. If users want to try smaller LLMs, they can edit the "run.py" to disable Vicuna-13b-v1.3 and enable another LLM instead.

For the basic test, users may need to change the paths in the artifact.

A.4 Evaluation workflow

A.4.1 Major Claims

Our paper has the following major claims:

(C1): *LLMs can effectively extract personal information from personal profiles. This corresponds to E1.*
(C2): *Existing defenses against PIE are not as effective as prompt injection as a defense. This corresponds to E2.*

A.4.2 Experiments

(E1): *[LLM PIE Extraction] [30 human-minutes + 20 compute-hour + 3GB disk]: this experiment evaluates how accurate LLMs can extract PII from profiles.*
How to: *Use official_code/run.py to run this experiments. The run.py is self-explanatory and contains configurable parts where users can enable/disable. After running run.py, users can use run_evaluate.py to generate evaluation results.*
Preparation: *The virtual environment described in above sections should be enabled. Depending on hardware, more packages may be required to be installed in the actual requirements. Please follow the error messages to install any required packages.*
Execution: *First, cd into official_code. Then, run python3 run.py*
Results: *The results will be in official_code/logs.*
(E2): *[Defense evaluation] [30 human-minutes + 15 compute-hour]: this experiment evaluates the defense effectiveness against LLM-based PIE.*
How to: *Similarly, users can use official_code/run.py to run this experiments by changing the value of the variable "defenses" to a list consisting of names of defenses. The run.py is self-explanatory and contains configurable parts where users can enable/disable. After running run.py, users can use run_evaluate.py to generate evaluation results.*
Preparation: *The virtual environment described in above sections should be enabled. Depending on hardware, more packages may be required to be installed in the actual requirements. Please follow the error messages to install any required packages.*
Execution: *First, cd into official_code. Then, adjust run.py to change the "defenses". Next, run python3 run.py*
Results: *The results will be in official_code/logs.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Enabling Low-Cost Secure Computing on Untrusted In-Memory Architectures

Sahar Ghoflsaz Ghinani, Jingyao Zhang, Elaheh Sadredini
University of California, Riverside
{sghof001, jzhan502, elahehs}@ucr.edu

A Artifact Appendix

A.1 Abstract

Our artifact contains all necessary source codes and scripts for evaluating our proposed security scheme. This paper leverages Multi-Party Computation (MPC) techniques, specifically arithmetic secret sharing, and Yao's garbled circuits, to securely outsource bandwidth-intensive computation to PIM. Additionally, we employ precomputation optimizations to prevent the CPU's portion of the MPC from becoming a bottleneck. We provided all the source codes and scripts for evaluating our scheme using UPMEM, the first publicly available PIM, over four data-intensive applications: Multilayer Perceptron inference (MLP), Deep Learning Recommendation Model inference (DLRM), linear regression training, and logistic regression training. This artifact allows researchers to reproduce our results, explore this area further, and expand our work. With this artifact, researchers can regenerate the performance results for UPMEM-Precompute-C(V), UPMEM-Runtime-C(V), and UPMEM-Runtime-A(2Y)-C(V), as presented in figures 13 to 18.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our research does not compromise user privacy or safety, and all data used in our experiments is either synthetic or randomly generated. Therefore, no Personally Identifiable Information (PII), sensitive data, or human subjects were involved.

A.2.2 How to access

The code is open-source and hosted on a GitHub public repository to facilitate reuse. Additionally, it is archived on Zenodo. We encourage other researchers to explore, reproduce, and modify our code, provided they give appropriate credit. We always recommend using the latest Zenodo release via the provided concept DOI. However, we have included the link to the initial version for consistency throughout our publication.

- Zenodo initial version: zenodo.org/records/14736864.
- Zenodo concept DOI: zenodo.org/records/14736863.
- GitHub: github.com/Secure-UPMEM/SecUPMEM.git.

A.2.3 Hardware dependencies

To evaluate our method, we utilize UPMEM PIM hardware, which primarily consists of standard DDR4-2400 DIMMs integrated with DPUs. Our setup includes 20 PIM-enabled DIMMs, providing a total of 160 GB of MRAM and 2560 DPUs working in parallel at a clock frequency of 350 MHz. The host server for the UPMEM system is equipped with a 2-socket Intel Xeon Silver 4110 CPU. To accurately reproduce our results, access to the actual hardware is necessary. UPMEM's PIM data centers are accessible upon request at `https://www.upmem.com/developer.`

A.2.4 Software dependencies

All the implemented applications require the UPMEM SDK, which can be installed based on the hardware specifications and is accessible at `https://sdk.upmem.com.` For dependencies related to MLP, DLRM, logistic regression, and linear regression, please refer to our baseline implementations as our work is built upon them.

- MLP (Link)
- DLRM (Link 1, Link 2)
- Logistic and Linear Regression (Link 1, Link 2)

A.2.5 Benchmarks

Our scheme is evaluated using MLP inference, DLRM inference, logistic regression training, and linear regression training. To evaluate our implementation, we use randomly generated inputs.

A.3 Set-up

A.3.1 Installation

To run this artifact on your local device, the UPMEM SDK must first be installed, which is available at `https://sdk.upmem.com`. However, as previously mentioned, reproducing our results requires access to real hardware (20 UPMEM PIMs) rather than the simulator. Once the UPMEM SDK is

installed, the artifact can be downloaded from GitHub and Zenodo.

A.3.2 Basic Test

Script *run_functionality.sh*, in the root directory, can be used to perform a simple functionality check. This file executes a basic test on all the applications sequentially and outputs execution time. To run a specific application, the *run_functionality.sh* script in the corresponding folder can be executed.

A.4 Evaluation workflow

A.4.1 Major Claims

Our major claim is as follows:

(C1): Compared to a secure CPU implementation, our framework achieves speedups of 14.66×, 9.80×, 2.64×, and 5.85× for MLP inference, DLRM inference, Linear Regression training, and Logistic Regression training, respectively. This is proven by the experiment (E1) described in Section 7.1 whose results are illustrated in Figures 14, 15, 16, 18.

A.4.2 Experiments

(E1): [30 human-minutes + 2.5 compute-hour + 32GB disk]:

How to: Our results can be reproduced by following the three steps below.

Preparation: Install the UPMEM SDK, as described in Section A.2.4, then clone our artifact.

Execution: Script *./run_reproduce.sh*, in the root directory, can be used to regenerate our results. This file executes all the applications with our configuration sequentially and outputs execution time for UPMEM-Precompute-C(V), UPMEM-Runtime-C(V), and UPMEM-Runtime-A(2Y)-C(V), as presented in Figures 13 to 18. To reproduce the results for a specific application, the *./run_reproduce.sh* script in the corresponding folder can be executed.

This script compiles and links the host and DPU source codes for a specific number of DPUs and tasklets. There is a Makefile for each application that facilitates the compilation and linking of the source codes.

Results: After running our script, the final execution time is reported. Table 1 explains the different notations used for manually determining execution times. The execution time is calculated using the formula provided below:

$$PIM\ Time = CPU - DPU + PIM\ kernel + DPU - CPU$$

$$Kernel\ time = Max[CPU\ Time, PIM\ Time]$$

$$Execution\ time = Kernel\ time + Merge + Verification$$

Table 1: List of timing notations

Timing notations	Explanation
$CPU - DPU$	CPU to DPUs transfer time
$DPU - CPU$	DPUs to CPU transfer time
$PIM\ kernel$	Kernel execution time on DPUs
$CPU\ Time$	Kernel execution time on CPU
$Kernel\ time$	The maximum of CPU Time and PIM Time
$Merge$	Merging time of CPU and PIM results
$Verification$	Verification time

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Software Availability Protection in Cyber-Physical Systems

Ao Li*, Jinwen Wang*, Ning Zhang
Computer Security and Privacy Laboratory
Washington University in St. Louis

A Artifact Appendix

A.1 Abstract

The artifact contains the source code of Gecko, an attack recovery approach that not only restores execution promptly after an attack but also disables exploited features to enhance system availability. It comprises three key components: (1) compartmentalization and instrumentation, (2) a shadow compartment mechanism, and (3) checkpoint/restore mechanisms. For this artifact evaluation, we demonstrate Gecko's functionality by showcasing its ability to recover ArduCopter from emulated attacks. Specifically, (1) ArduCopter is compartmentalized using our customized compilers, and (2) it recovers from a triggered crash with the support of the shadow compartment mechanism and checkpoint/restore mechanisms. We aim to streamline the Artifact Evaluation (AE) process by providing a pre-configured machine with all necessary dependencies.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Conducting the Artifact Evaluation (AE) for Gecko does not raise any security, privacy, or ethical concerns. All AE tasks are performed on a dedicated remote machine specifically used for AE, ensuring that the process remains isolated and does not interact with the reviewer's personal or sensitive code/data.

A.2.2 How to access

The source code is available on GitHub at https://github.com/WUSTL-CSPL/Gecko, as well as on Zenodo: https://zenodo.org/records/15049225.

A.2.3 Hardware dependencies

There is no specific hardware dependency. For convenience during evaluation, we provide access to a remote machine set up via TeamViewer.

A.2.4 Software dependencies

Gecko is evaluated using ArduPilot (Copter-4.0) as the target application. The testing environment consists of Ubuntu 20.04 as the operating system and LLVM version 13.0.1 as the compiler.

A.2.5 Benchmarks

None

A.3 Set-up

This setup is intended for users who wish to build the system from scratch. However, to simplify the process, we provide a pre-configured Docker container that allows reviewers to skip Section 3 entirely. The container is available at: ghcr.io/a01ixxx/gecko-image:stable.

A.3.1 Installation

Prerequisite APT packages:

```
$ sudo apt install cmake build-essential
    make texinfo bison flex ninja-build git
    gitk git-gui ncurses-dev texlive-full
    binutils-dev python3-networkx python3-
    matplotlib python3-pygraphviz python3-
    serial python3-pip python3-distutils
    python-is-python3 tmux libgtk-3-dev
    libwebkit2gtk-4.0-dev libjpeg-dev
    libtiff-dev libsdl1.2-dev libgstreamer1
    .0-dev
```

Prerequisite pip packages:

```
pip install community pydot opencv-python
    future MAVProxy wxPython
```

LLVM Installation: Gecko is tested on LLVM-13. User can either install the package on ubuntu or compile from source code. Due to space constraints, please refer to the official instructions for more details.

Build SVF program analysis tool:

```
$ cd /path/to/Gecko/SVF
$ ./build.sh
```

Build CRIU checkpoint/restore tool:

```
# install depdendencies
$ sudo apt install libdrm-dev gnutls-dev
    libnftables-dev libbsd-dev  libprotobuf-
    dev libprotobuf-c-dev protobuf-c-
    compiler protobuf-compiler python3-
    protobuf libnl-3-dev libcap-dev  uuid-
    dev libbsd-dev libnftables-dev libnet1-
    dev gnutls-dev libdrm-dev
$ cd /path/to/Gecko/checkpoint_restore
$ make clean
$ make
```

ArduPilot Installation:

```
$ cd /path/to/Gecko/ardupilot
$ ./waf configure --board sitl
$ ./waf build
```

A.3.2 Basic Test

You can verify the success of the LLVM installation:

```
$ llvm-config --version
13.0.1
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Gecko is capable of compartmentalizing and instrumenting the system while enabling real-time recovery.
(C2): Gecko's recovery capability allows CPS to recover from triggered failures.

A.4.2 Experiments

(E1): This verifies the compartmentalization and instrumentation processes (C1), which are applied at compile time. [5 human-minutes + 15 compute-minutes]

Given CPS software such as ArduPilot, Gecko uses command line instructions to automatically compartmentalize and instrument the software.

```
$ cd ~/ardupilot_redcaps/
$ ./1_compartmentalization.sh
```

Result E1-1: The compartmentalization result is in the file *./build/sitl/compartments_result.json*. The functions and variables are partitioned into different compartments (code and data regions).

Next, Gecko uses the following command line to automatically instrument the defense mechanism in the given application with our customized compiler.

```
$ ./2_instrumentation.sh
```

Result E1-2: The final binary result is the file *./build/sitl/bin/arducopter*. You can use the following command to check the instrumented attack detection codes.

```
objdump -D ./build/sitl/bin/arducopter |
    grep _dfi
```

(E2): This demonstrates Gecko's ability to recover from a crash (C2). [3 human-minutes + 10 compute-minutes]

Launching the ArduPilot simulation requires multiple commands, so we provide three scripts to simplify the process.

The first script launches ArduPilot and checkpoints a program in memory, which will be used later.

```
$ cd ~/ardupilot_recovery
$ ./prepare_checkpoint.sh
# You may need to enter the sudo passwd
```

The second script automatically configures the drone, launches the mission, and opens two panels to display simulation information.

```
$ cd ~/ardupilot_redcaps/
$ ./launch_demo.sh
[Gecko] Using the native block.
[Gecko] Using the native block.
```

Please wait 3-5 minutes for the drone to start executing the mission, as shown in the figure below:

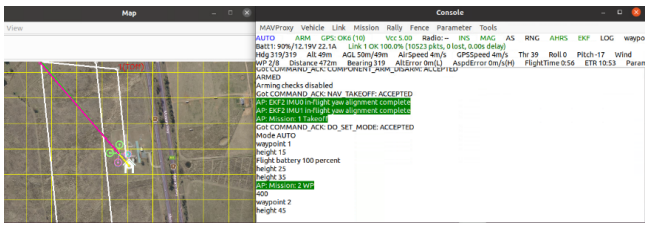


Figure 1: ArduCopter mission example.

Then, return to your main terminal by detaching from the current tmux session: press `Ctrl+b`, then `d`. Next, you can run the thrid script, which is designed to trigger a program crash and then recover it from a checkpoint.

```
$ ./launch_attack_recovery.sh
```

Result E2-1: This will cause the program, namely *arducopter*, to crash and then restore it from a previously saved checkpoint. After recovery, the system transitions to a shadow compartment. This is indicated by continuous logs showing the use of the recovery block, as shown below:

```
[Gecko] The task is restored !!!
With response time: xxx seconds
[Gecko] Using the recovery block.
[Gecko] Using the recovery block.
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: GDMA: Fully Automated DMA Rehosting via Iterative Type Overlays

Tobias Scharnowski, Simeon Hoffmann, Moritz Bley, Simon Wörner, Daniel Klischies[1], Felix Buchmann, Nils Ole Tippenhauer, Thorsten Holz, and Marius Muench[2]

CISPA Helmholtz Center for Information Security
[1]Ruhr-Universität Bochum, [2]University of Birmingham

A Artifact Appendix

A.1 Abstract

This artifact contains the information necessary to reproduce all claims from the paper *GDMA: Fully Automated DMA Rehosting via Iterative Type Overlays*.

A.2 Description & Requirements

This artifact is a fuzzing artifact and hence scales better the more resources it can use. In our setup, we ran each experiment 10 times, 24 hours each. We used two Intel Xeon Gold 5320 CPUs (26 physical cores and 2.20GHz each), 256 GB of RAM, and 1TB SSD storage.

For artifact evaluation, we recommend running the experiment with 26 cores, 3 times, 12 hours each. This ensures that even on moderate hardware, the artifact evaluation time period suffices. With this configuration, one full pipeline run takes eleven days. Please take this into account when inspecting this artifact. However, it is possible to split up the long-running parts over several machines to reduce runtime.

A.2.1 Requirements

All experiments were run on an Ubuntu 22.04 machine. We provide reduced fuzzing configurations to account for artifact reviewers without access to hardware clusters. However, our recommended configuration still requires eleven days of computing on a single piece of minimal hardware. Therefore, please plan accordingly.

A.2.2 Security, Privacy, and Ethical Concerns

It may happen that new bugs are found during the evaluation of this artifact. In that case, please report them responsibly.

A.2.3 How to Access

Please obtain all required files:

(A1) GDMA artifact: `https://github.com/fuzzware-fuzzer/fuzzware/tree/DMA`
(A2) the GDMA experiments artifact `https://github.com/fuzzware-fuzzer/gdma-experiments`
(A3) the DICE / P2IM reproduction archive: `https://doi.org/10.5281/zenodo.15600641` (`dice_docker.tar.zstd`)

Stable URL The stable URL of the artifact is the following: `https://doi.org/10.5281/zenodo.15600641`.

A.2.4 Hardware Dependencies

This artifact is a fuzzing artifact and hence scales better the more resources it can use. In our setup, we ran each experiment 10 times, 24 hours each. We used two Intel Xeon Gold 5320 CPUs (26 physical cores and 2.20GHz each), 256 GB of RAM, and SSD storage.

The minimal requirements are 26 cores, 64GB RAM and 500GB disk storage. More cores can speed up the experiments, as can multiple machines.

A.2.5 Software Dependencies

Please install the following software dependencies:

```
docker curl ca-certificates patchelf texinfo
automake gcc zstd pip zip python3-venv
binutils-arm-none-eabi
```

Also, make sure that you are in the docker group that you can use docker without elevated permissions[1].

A.2.6 Benchmarks

None.

A.3 Setup

The setup is tested on Ubuntu 22.04.

[1]https://docs.docker.com/engine/install/linux-postinstall/

A.3.1 Installation

Installation of GDMA. Start by cloning the GDMA artifact and installing it (A1):

```
git clone --recurse-submodules -b DMA
https://github.com/fuzzware-fuzzer/fuzzware.git;

cd fuzzware; ./build_docker.sh
```

This performs a regular docker build. If this build completes the `GDMA` installation was successful.

Installation of GDMA experiments. Obtain the GDMA experiments artifact (A2):

```
git clone https://github.com/fuzzware-fuzzer/
gdma-experiments.git
```

Enter the scripts directory and install the requirements in a clean virtual environment.

```
cd gdma-experiments/scripts;
python3 -m venv venv && source venv/bin/activate &&
pip install -r requirements.txt
```

We expect all further commands to be executed from within the virtual environment. To finish the installation of the GDMA experiments artifact, move the three remaining archives (A3-A5) to their place as described below.

Setup IDEs. Some samples need their integrated development environments (IDEs) to successfully build. To obtain the required IDE files, follow the instructions at `gdma-experiments/src/02-extended-test-suite/targets/CY8CKIT-062-BLE/` and at `src/02-extended-test-suite/targets/LPC1837-mmio`. Move the resulting files into the `ides` directory.

Setup DICE / P2IM reproduction containers. Move the DICE / P2IM reproduction archive to `gdma-experiments/docker-saves`. This requires no unpacking:

```
mv dice_docker.tar.zstd
gdma-experiments/docker-saves/
```

Configuring the system for fuzzing. Execute the following commands to prepare the system for fuzzing with afl and fuzzware.

```
echo 512 > /proc/sys/fs/inotify/max_user_instances;
echo 524288 > /proc/sys/fs/inotify/max_user_watches;
echo core >/proc/sys/kernel/core_pattern;
cd /sys/devices/system/cpu;
echo performance | tee cpu*/cpufreq/scaling_governor
```

A.3.2 Basic Test

To test that GDMA works correctly, execute:

```
fuzzware/run_docker.sh;
fuzzware pipeline --help
```

This gives you a shell in a docker container and executes the fuzzware help. If `--dma` is among the listed options, then GDMA is set up correctly.

To verify that the eval scripts are set up correctly, execute

```
cd gdma-experiments/scripts && python cli.py
```

If this shows you an argparse-style overview of command-line options, then the scripts are setup correctly.

To confirm progress during the evaluation, inspect the folders of each experiment (e.g., `02-extended-test-suite/` or `01-dice-comparison/02-real-firmware/`). If it contains an output folder, then the fuzzer has at least started there. If the next experiment also contains an output folder, then the fuzzer has completed the previous experiment.

A.4 Evaluation Workflow

A.4.1 Major claims

In the paper, we make the following four major claims:

(M1) GDMA correctly models all Direct Memory Access (DMA) types from state of the art data sets.
(M2) GDMA provides coverage for 6x the DMA mechanisms compared to the state of the art.
(M3) GDMA improves coverage on real-world targets.
(M4) GDMA identifies 6 previously unknown bugs.
(M5) GDMA has a low false positive rate.

A.4.2 Open Science

In addition, this artifact contains the following 5 components in accordance to the open science policy.

(O1) The diverse dataset from Table 2.
(O2) The example applications firmware dataset from Figure 6.
(O3) Ground truth metadata for the DICE dataset.
(O4) A docker-based reproduction environment for DICE and P2IM.
(O5) The source code of GDMA.

A.4.3 Experiments

To show our major claims and open science contribution, we provide the following experiments (eleven days runtime on recommended machine):

(E1): Diverse data set reproduction: rebuild the diverse dataset from its sources (10 human minutes + 0.5 CPU hours + 45GB disk space (if docker export or import is used), O1)
(E2): Example applications reproduction: rebuild the example applications from its sources (10 human minutes + 0.5 CPU hours + 45GB disk space (if docker export or import is used), O2)

(E4): P2IM / DICE containerized reproduction environment: A dockerized environment to build both P2IM and DICE , and run each tool (10 human-minutes + 12h on recommended machine + 20GB disk space, O4)

(E5-1): Dice Unit tests: Comparison to state of the art on their unit test dataset (10 human-minutes + 48h on recommended machine + 40GB disk space, M1)

(E5-2): Dice Fuzzing tests: Comparison to state of the art on their fuzzing dataset (10 human minutes + 24h on recommended machine + 70GB disk space, M1)

(E6): Diverse dataset performance: Evaluation of GDMA on the diverse dataset (10 human minutes + 48h on recommended machine + 30GB disk space, M2)

(E7): Example applications performance: Evaluation of GDMA on the example applications dataset (10 human minutes + 48h on recommended machine + 70GB disk space, M3)

(E8): Finding new bugs: Evaluate if GDMA can be used to find new bugs (10 human minutes + 24h on recommended machine + 70GB disk space, M4)

(E9): False positives: Evaluate if GDMA finds false positives in P2IM unit test set or the diverse dataset (10 human minutes + 72h on recommended machine + 30GB disk space, M5)

We split the reproduction into three parts: build target reproduction, DICE / P2IM execution environment test, and GDMA evaluation. As the first step, enter the `scripts/` directory: `cd gdma-experiments/scripts`. The remainder of this section assumes that all commands are executed from within this directory.

A.4.4 Reproducing O3

We do not provide an experiment script to reproduce **O3**. Extraction of ground truth metadata was a one-time manual effort, as it involved reverse-engineering the provided firmware samples. Please find the results for DICE unit tests in `ground_truth_dice_perspective.yml`[2]. Please find DICE fuzzing test ground truth metadata in their respective directory, in the `config_generic_dma_manual.yml`[3] file.

A.4.5 Build Target Reproduction (E1-E2)

The first section of the artifact reproduction describes how to rebuild the diverse dataset and the example applications that we provide as part of the artifact.

Diverse Dataset (E1). To test the reproduction of the diverse dataset, execute the following commands:

b) Rebuild from scratch: Run

```
python cli.py build-2
```

This builds docker containers from scratch, importing the IDEs from `ides`.

After successfully rebuilding the targets, execute

```
python cli.py update-2 -pd
```

, which checks for each binary if the newly-built version differs from the existing one.

Example Applications (E2). The reproduction for the example applications follows the same idea. Execute

```
python cli.py build-3
```

for a rebuild from scratch. You can verify that all binaries built successfully, as above, by executing

```
python cli.py update-3 -pd
```

, or by removing them before the build process and counting them afterwards (there should be 10 binaries).

Using the new binaries. You can use the samples you built in the previous steps to perform the evaluation. However, to keep the overhead for you as low as possible we recommend to use the binaries we provide. IF you want to entirely verify that the evaluation works with the rebuilt binaries you can update the diverse dataset binaries and the example applications with the following commands

```
python cli.py update-2 -u;
python cli.py update-3 -u
```

. Note that this might invalidate some of the present configurations as some targets require manual tweaking of the memory maps. We advise you to first perform the evaluation with the dataset in the archive and to test out the update mechanism afterwards if you are interested to further investigate this aspect of the artifact.

A.4.6 DICE / P2im Execution Environment Test (E4)

This part builds and runs both DICE and P2IM, which is related work that GDMA was evaluated against. The targets for this part of the evaluation stem from the DICE evaluation and will hence be call DICE fuzzing tests.

This comes in two flavors: one rebuilds DICE and P2IM from scratch:

```
python cli.py run-dice
```

and creates an evaluation-ready docker container, the other imports existing docker images from `docker-saves`:

```
python cli.py run-dice -d
```

Both run DICE and P2IM on the DICE fuzzing tests afterwards.

You can modify fuzzing campaign parameters by editing `01-dice-comparison/02-real-firmware/01-evaluation/config.txt`. We provide wrapper scripts to directly execute DICE reruns from the `scripts/` directory. On the recommended machine, execute:

[2] 01-dice-comparison/DICE-results/ground_truth_dice_perspective.yml

[3] 01-dice-comparison/02-real-firmware/firmware/GPS-Receiver/config_generic_dma_manual.yml

```
./dice_12h_3runs.sh
```

Investigating the results. To investigate the results navigate to:

```
cd gdma-experiments/scripts/results/p2im_dice_results
```

This directory contains two subdirectories: `DICEFuzzBase` and `P2IMFuzzBase`, which contain the data for the respective platform. The structure of each of these subdirectories is `<sample>/<run>/`. To investigate the results of P2IM's/DICE's modeling and fuzzing please refer to the respective projects. In addition to the various peripheral modeling and fuzzing-related directories each run contains a `coverage` folder. In there is a file that contains the total number of basic blocks covered (`bbl_cnt`) and a file that contains the mapping of inputs to the basic blocks they cover (`bbl_cov`). Further each fuzzing run contains an `out.txt` file, that contains a mapping between fuzzing time and total number of basic blocks covered at that time. If a directory does not contain an `out.txt` file please refer to the troubleshooting section.

To verify, whether the results of the fuzzing run match those shown in the paper you can inspect the different `out.txt` files and compare the achieved number of basic blocks with Figure 7 in the paper.

Troubleshooting. Unfortunately, in some cases the coverage and plotting information can not be generated for all targets even though the fuzzing worked fine. This usually manifests itself with error messages such as:

```
cp: cannot stat 'p2im_dice_results/DICEFuzzBase/
Modbus/2/out.txt': No such file or directory
```

To verify whether this occurred during your evaluation you can perform the following steps:

```
cd gdma-experiments/scripts
find ./results/p2im_dice_results -name "out.txt"
| wc -l
```

The resulting number should be `11 * <runs>`, where <runs> is the number of repetitions specified through the run script (e.g., `dice_12h_3runs.sh` → <runs>=3). This "formula" is derived by multiplying the number of targets (6) by 2 (each is fuzzed twice: once with DICE and once with P2IM) and then subtracting the number of repetitions once, as one sample (MIDI-Synthesizer) does not work with P2IM, therefore, no output file is created.

If you notice that the number of generated `out.txt` files is lower than expected, we provide a script to rerun the coverage generation, based on the data from the fuzzing runs.

To this end, you can run the following commands:

```
cd gdma-experiments/scripts
cd ../01-dice-comparison/02-real-firmware/
04-coverage-only/
./rerun_cov.sh ../../../scripts/results/
p2im_dice_results/../02-results
```

Afterward, you can do the same verification from above to check whether the issue was fixed.

A.4.7 GDMA Evaluation (E5-E9)

This part reruns all 5 fuzzing campaigns and collects the results in the `scripts/results/` directory. To start the GDMA evaluation immediately with default parameters, move to the top-level directory `gdma-experiments` and execute:

```
./eval-recommended.sh
```

Optional Configuration. The evaluation setup is configurable to adapt to different availability of computing power. We provide configurations for a range of computing power. We recommend using one of the wrapper scripts to keep the manual configuration effort low. Execute one of the wrapper scripts (e.g. `rerun_experiments_12h_3runs.sh`) to use these preset configurations.

If your hardware setup is incompatible with the provided presets, you can configure the parameters of the fuzzing campaigns in the configuration file `scripts/.experiments-config.yml`. It contains 4 keys: `experiments` defines which of the 5 fuzzing campaigns you want to rerun, `cores-per-experiment` defines the number of cores you assign for the fuzzing campaigns, `fuzzing-time` the timeout for the fuzzer and `runs-per-firmware` the number of runs of each target.

Running the evaluation. If you want to perform the evaluation on recommended hardware and settings, execute:

```
./eval-recommended.sh
```

in `gdma-experiments` directory root. You can run the full evaluation with preselected parameters by executing one of the provided wrapper scripts (e.g. `rerun_experiments_12h_3runs.sh`). The scripts run all fuzzing campaigns with the parameters indicated in the wrapper name. Afterwards, it performs all evaluations, placing the results in the `scripts/results/` directory.

If you have similar or better hardware resources than those used by us in our evaluation (see A.2.4), we you can run the full evaluation as ran by us. To do that, enter the `scripts` directory and execute the script:

```
rerun_experiments_24h_10runs.sh
```

However, we understand that not everybody has access to such resources. If this is the case for you, we recommend to execute

```
rerun_experiments_12h_3runs.sh
```

from inside the `scripts` directory. This requires less than 25% of compute time, and yields comparable results. Note that the x-axis (the time scale) of the resulting plots will differ. Edit and use `python estimate_runtime.py` to estimate the runtime for a given number of execution time, runs and cores.

Investigating the results. The `scripts/results` directory contains all evaluation artifacts.

01_unit.txt (E5-1). This contains an ASCII version of Table 7 in the paper.

02_fuzzing.pdf (E5-2). This contains the coverage plots of the DICE fuzzing targets (Figure 7 in the paper). If you did not run the DICE / P2IM environment test (see A.4.2), the data for plotting DICE and P2IM is equal to that used in the paper. Running the DICE / P2IM environment test automatically overwrites the old data with the latest results.

02.txt (E6). This file contains information about password characters found in the diverse dataset. It marks a target as successfully executed if some password character were found. It requires 1 identified password character if a single DMA buffer was involved, or 2 password characters (1 per buffer) if two buffers were involved.

02-fp.txt (E9). This is the result file of diverse dataset false-positive analysis. This analysis compares the DMA configurations identified by GDMA to a manually constructed ground truth and reports differences.

03-cov.pdf (E7). The coverage plots for the example application dataset (Figure 6 in the paper).

04.txt (E8). This experiment replays all reproducer inputs to verify that they indeed crash the target. If you want to verify the CVEs found by GDMA, you need to manually inspect the crashes that your run found. Please find the raw output of the fuzzing campaign in `04-finding-new-bugs/output/`. Note that the crashes are not necessarily triggering the identified CVEs. Also, note that depending on your execution time, some bugs may not be found, as the time to crash is high for some bugs.

05.txt (E9). This experiment investigates if `dma_config.yml` files were created for any of the P2IM unit tests. The existence of any `dma_config.yml` is a false positive.

A.5 Notes on Reusability

Our artifact uses a python script as main tool to rerun our experiments. This section contains information on how to use all parts individually. Each of the following commands is invoked as

```
python cli.py <headline>
# e.g. python cli.py run-experiments
```

`python cli.py --help` gives a brief overview over each command, and each subcommand also has an individual `--help` output.

A.5.1 Running the Campaigns

run-experiments. Runs the actual fuzzing campaigns. Is configured via the `.experiments-config.yml` file. This file controls the parameters of the fuzzing run. Use the `-r` flag to actively select the configuration, otherwise the default configurations in the directories of the fuzzing run will be used. Use `-d` to perform a dry-run: it prints all the commands instead of running them.

A.5.2 Evaluating the Campaigns

eval-1-1. Builds the unit test table (Table 7 in the paper). By default, builds a human readable table (ASCII art style). Pass `--latex` to get latex code for the table (this is also used in the paper).

eval-1-2. Builds the plot for the DICE Fuzzing targets (Figure 7 in the paper). You can pass directories that contain the output from an experiment run via command line flags (use `--help` to see which ones) relative to the repository root. By default, it uses paths as present in the repository structure. Use `--runtime` and `--num-runs` to set the correct runtime and number of runs per target.

eval-2. Evaluates the functionality of the DMA emulation mechanism. This takes the results of a fuzzing campaign on the diverse data set and confirms that GDMA found password characters. If there was a single consecutive buffer in memory, the evaluation checks for the discovery of the first password character. If the DMA mechanism uses two buffers, the evaluation checks whether GDMA found password characters for both buffers. Use `--results` to pass an alternative path to the output of `run-experiments`.

eval-2-fp. Check that GDMA found the correct DMA configuration during its run. We compare the Memory-mapped I/O (MMIO) register and the written buffer pointer to compare configurations. Note that some targets have a non-deterministic way of providing a DMA channel, and consequently, multiple MMIO registers are correct. `--data` points to the results of the fuzzing campaign, `--groundtruth-root` points to a directory that contains the reference configurations.

eval-3. Build the coverage plots for the real-world targets (E3, Figure 6 in the paper). Interface is similar to **eval-1-2**.

eval-4. Rerun the bug reproducers found in the repository and investigate if they crash. Pass target directory with `--targets`.

eval-4-hooks. Rerun the reproducers found in the repository with a special configuration that prints *Heureka* when found. If you pass `-f`, a new fuzzing campaign is started that prints *Heureka* when the bugs are triggered.

eval-5. False positive analysis. Search for `dma_config.yml` in the results directory of experiment 5.

A.5.3 Rebuilding the Targets

build-2. Rebuild targets for E1. This comes in two flavors: the default is a full rebuild from scratch. This requires the presence of several proprietary IDEs which we cannot provide in the final repository (after Artifact evaluation) due to legal reasons. Please see the `readme.md` files of the individual targets

for details on how to obtain them via the official channels.The full rebuild builds docker images and runs the build process in docker, moving the final build binaries in the respective `output` folder. If you pass `-e`, then the docker image will be saved for later reuse. If you pass `-d`, docker images will be loaded from the `docker-saves` folder in the repository root (assuming you exported them before). These docker saves contain the IDE setups and will build binaries during the run.

update-2. Check if the binaries found in the `output` folders of the dataset for E2 match the ones used in the evaluation. If you pass `-pd`, a diff is printed. If you pass `-u`, binaries that are not identical will be updated (and also get a new `config_autogen.yml`).

build-3. `build-2`, but for E2 targets.

update-3. `update-2`, but for E2 targets.

run-dice. Build P2IM and DICE , the related work tools that we evaluate against. Similar to the interface of `build-2`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Kintsugi: Secure Hotpatching for Code-Shadowing Real-Time Embedded Systems

Philipp Mackensen[*], Christian Niesler[†], Roberto Blanco[‡], Lucas Davi[†], Veelasha Moonsamy[*]
[*]Ruhr University Bochum, [†]University of Duisburg-Essen, [‡]Eindhoven University of Technology/MPI-SP

A Artifact Appendix

A.1 Abstract

This artifact appendix provides the source code for the prototype implementation of KINTSUGI, all source codes for the experiments described in the paper, and a Dockerfile that enables the creation of an image containing all the prerequisites necessary for successfully evaluating KINTSUGI. For each performance experiment, we provide the measurement files used as source for the tables and plots in the paper's evaluation. Additionally, for each of the 10 real-world CVEs and the security experiments, we provide logs reflecting the expected outputs. We made our artifact openly accessible and focused on an easy-to-evaluate solution. Once the Docker image is built, one can run each experiment using the provided scripts.

A.2 Description & Requirements

To run all artifacts, you must have an nRF52840-DK board, which has been used to test all experiments. All experiments were conducted (building/flashing/evaluating) on a host machine running Kubuntu 22.04.4 LTS x86/x64, featuring a 12th Gen Intel i7-1206P (16) processor. We provide a docker file that contains all the necessary commands to successfully build an environment that allows to build, flash, debug and evaluate KINTSUGI.

A.2.1 Security, privacy, and ethical concerns

To the best of our knowledge, there are no security risks associated with running our artifacts on the host machine of any evaluators. However, we must note that the Docker is run with `--privileged` due to the need to flash the device's firmware.

A.2.2 How to access

The artifacts are publicly available at https://doi.org/10.5281/zenodo.15592036.

A.2.3 Hardware dependencies

Nordic *NRF52840-DK* development board: Required to run the firmware and reproduce the core results.

Host machine (Linux): Used to build and flash firmware, interface with the *NRF52840-DK*, collect results and output tables / plots. A standard desktop or laptop with $\approx 8GB$ of RAM and $\approx 64GB$ of storage, with USB and internet access is sufficient.

A.2.4 Software dependencies

We recommend using Kubuntu 22.04.4 LTS, as all experiments have been successfully tested on this distribution. We have also verified that the experiments work on a virtual machine running Ubuntu 22.04 LTS. Additionally, it is necessary to have bash installed (which comes standard with Linux) and Docker, which will handle the installation of all other prerequisites.

A.2.5 Benchmarks

None.

A.3 Set-up

We provide a Dockerfile that allows to build a docker image in which everything to perform our experimental evaluation is gets installed.

A.3.1 Installation

To begin, install the Docker engine on Linux: https://docs.docker.com/engine/install/. Next, download the KINTSUGI artifact, extract it, and run the `build.sh` script to set up the Docker image. Building the Docker image will take approximately 30 minutes and it will occupy approximately $28GB$ of storage.

A.3.2 Basic Test

To verify that Docker can successfully build every experiment, please run the `verify.sh` script. This script will check that

all experiments are built correctly; if the Docker image is missing, it will build it as well. For each experiment, the script will output whether the build was successful or not. It takes approximately 8 min to verify everything.

Note, that this test does not require the existence of the hardware as this is a pure software test.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): KINTSUGI introduces minimal performance overhead across the hotpatching process. Validation and scheduling have consistent measurement times, independent of hotpatch sizes. This was proven by experiment (**E1**) and explained in *Section 6.1*, "Manager" (Table 2), for hotpatch sizes based on `RapidPatch` and `AutoPatch`. An extended version is provided in *Appendix D* (*Table 5*), and hotpatch sizes from KINTSUGI are discussed in *Section 6.4*, with results in *Appendix E* (*Table 6*).

(C2): The performance overhead introduced by KINTSUGI into the context switch of an RTOS is minimal. This is proven by experiment (**E2**) and described in *Section 6.1*, "Guard & Applicator" (*Table 3*).

(C3): Successively processing and applying hotpatches scales linearly with the number of hotpatches, as demonstrated with up to 64 hotpatches. This was proven by experiment (**E3**) and is described in *Section 6.2*, with the results shown in *Figure 5*.

(C4): The memory overhead introduced by KINTSUGI grows predictably with the maximum hotpatch size and slot count. The overhead ranges from an average of $3.5KB$ to $42KB$. This is proven by experiment (**E4**) and described in *Section 6.3*, with the shown in *Figure 6*.

(C5): KINTSUGI is capable of hotpatch real-world vulnerabilities as we demonstrated with 10 CVEs. This is proven by experiment (**E5**) and is described in *Section 6.4*. Resulting hotpatch sizes are shown in *Table 4*.

(C6): KINTSUGI is able to prevent tampering attacks from an adversary before, during and after the hotpatching process. This is proven by experiment (**E6**) and is described in *Section 8*.

A.4.2 Experiments

We provide evaluation scripts in the directory `evaluation_scripts` which are interactive and include descriptions of their working and how to correctly use them. If a script automatically produces outputs (i.e., the performance experiments) they will be stored in the folder `measurement_results` under their corresponding subdirectories. We clearly mark experiments that do *not* automatically produce outputs. Those experiments require access to the UART of the board through, e.g., `screen`, `minicom` or `puTTY` with a baudrate of 115200.

(E1): [`micro-benchmarks.sh`] [$\approx$ 6 computer-hours]
Execute the *Micro-Benchmarking* experiments of the *Manager* to prove (**C1**), measuring how each component inside of the *Manager* takes to execute.
Automatic Output: *Yes.*
Execution: The measurement does not require user interaction.
Results: Upon execution, the user will be asked to select the hotpatch sizes to measure. They can choose between *RapidPatch & AutoPatch* (Section 6.1) or *Kintsugi* (Section 6.4). Measuring a single configuration takes approximately 100 seconds, hence why the script will stay at "*Reading UART output*" for a while. All results will be stored in `micro-benchmarks/manager` with raw measurements being stored in `measurements` and the resulting tables in `output`. The measurements in the table should be as close as possible to those claimed in (**C1**).

(E2): [`context-switch.sh`] [$\approx$ 2 computer-minutes]
Execute the *Micro-Benchmarking* experiments of the *Guard & Applicator* focusing on the *Context-Switch* of the RTOS to prove (**C2**), demonstrating how much time KINTSUGI adds to the context-switch of the RTOS.
Automatic Output: *Yes.*
Execution: The measurement does not require user interaction.
Results: All results will be stored in `micro-benchmarks/context-switch` with raw measurements being stored in `measurements` and the resulting tables in `output`. The measurements in the table should be as close as possible to those claimed in (**C2**).

(E3): [`scalability.sh`] [$\approx$ 4 computer-hours]
Execute the *Hotpatching Scalability* experiments to prove (**C3**), measuring KINTSUGI's performance when applying multiple consecutive hotpatches, demonstrating that it grows linearly in the number of hotpatches.
Automatic Output: *Yes.*
Execution: The measurement does not require user interaction.
Results: All results will be stored in `scalability` with raw measurements being stored in `measurements` and the resulting plot will be stored in `output`. The plot should show linear growth in the dimension of number of hotpatches and be as close as possible to the results claimed in (**C3**).

(E4): [`resource-utilization.sh`] [$\approx$ 3 computer-hours]
Execute the *Resource Utilization / Memory Overhead* experiments to prove (**C4**), measuring the memory overhead introduced by KINTSUGI under different parameter configurations of hotpatch counts and sizes.
Automatic Output: *Yes.*
Execution: The measurement does not require user interaction.

Results: All results will be stored in `resource-utilization` with raw measurements being stored in `measurements` and the resulting plot will be stored in `output`. The plot should show an increase in memory overhead in both dimensions and be as close as possible to the results claimed in (**C4**).

(E5): [`realworld-cves.sh`] [≈ 30 minutes total]
Perform the *Real-World Hotpatching* experiments to prove (**C5**), demonstrating that KINTSUGI is capable in hotpatching real-world CVEs.
Automatic Output: *No.*
Preparation: Connect to the device's UART as described above. Typically the access will be over `ttyACM0` or `ttyACM1`.
Execution: Upon executing the script the user will be asked to input a number between 1 and 10 to decide which CVE to execute on the board.
Results: The results differ for each CVE experiment. We provide details about expected outputs in `experiments/realworld_cves/README.md`. This file also reflects all the outputs that we have obtained during our experimental evaluation.

(E6): [`security.sh`] [≈ 10 minutes total]
Perform the *Security* experiments to prove (**C6**), demonstrating that KINTSUGI is resistant against adversaries *before*, *during* and *after* the hotpatching process.
Automatic Output: *No.*
Preparation: Connect to the device's UART as described above. Typically the access will be over `ttyACM0` or `ttyACM1`.
Execution: Upon executing the script the user will be asked to input a number between 1 and 3 to decide which of the three security experiments to run.
Results: The results differ for each security experiment. We provide details about expected outputs in `experiments/security/README.md`. This file also reflects all the outputs that we have obtained during our experimental evaluation.

A.5 Notes on Reusability

We provide additional details on how to properly integrate the tool into the *Zephyr* and *FreeRTOS* RTOSes in the `example_rtos_integration` folder. We provide the necessary files that must be adapted to the respective RTOS and instructions on how to do so manually. In our Docker, we require patches to modify the corresponding Git repositories. Therefore, the git-patches for the RTOSes can potentially be used immediately.

Integrating KINTSUGI into an RTOS should be straightforward. To do this, create a task for the *Manager* and modify the context switch to check if the *previous* or *next* task is the *Manager*. Additionally, call the *Guard & Applicator* before restoring the execution context of the next task.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Dumbo-MPC: Efficient Fully Asynchronous MPC with Optimal Resilience

Yuan Su
Xi'an Jiatong University

Yuan Lu
Institute of Software
Chinese Academy of Sciences

Jiliang Li
Xi'an Jiatong University

Yuyi Wang
CRRC Zhuzhou Institute

Chengyi Dong
Xi'an Jiatong University

Qiang Tang
The University of Sydeny

A Artifact Appendix

This repository contains the artifact for the USENIX 2025 submission "Dumbo-MPC: Efficient Fully Asynchronous MPC with Optimal Resilience".

A.1 Abstract

Fully asynchronous multi-party computation (MPC) has superior robustness in realizing privacy and guaranteed output delivery (G.O.D.) against asynchronous adversaries that can arbitrarily delay communications. We design a concretely efficient fully asynchronous MPC—Dumbo-MPC with entire G.O.D. and optimal resilience against $t < n/3$ corruptions (where n is the total number of parties). The codebase includes the implementation for Dumbo-MPC.

Our artifact is built with Ubuntu 20.04 LTS, and can be run on a single machine with multi-threaded or multi-processes emulation, or in a distributed setting consisting of multiple amazon-web service (AWS) virtual machines. There are two important parameter choices of our artifact: (i) num_node: number of nodes in the MPC protocol, (ii) batchsize: the number of secrets to be shared.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no concerns when running this artifact locally. Please note that executing experiments on your AWS infrastructure involves the creation of multiple EC2 instances, resulting in associated costs. Please manually check that any created machines are terminated afterward.

A.2.2 How to access

All of our files are publicly accessible and can be downloaded from: `https://github.com/dcy456/Dumbo-MPC`, which is also at the permanent Zenodo repository `https://zenodo.org/records/15123146`.

A.2.3 Hardware dependencies

Our code can be executed at the local machine installing Ubuntu 20.04 LTS, or among a few distributed AWS instances. For distributed evaluations, EC2 instances of type c6a.8xlarge with 32 vCPUs and 64 GB memory are recommended for reproducing the evaluation results reported in the paper.

A.2.4 Software dependencies

The following dependencies are required:

- Operating system: Ubuntu 20.04 LTS.
- System dependencies: `make bison flex libgmp-dev libmpc-dev libntl-dev libflint-dev python3 python3-dev python3-pip libssl-dev wget git build-essential curl tmux`. We require specific versions of programming languages: Python 3.8.x, Rust 1.86.0-nightly and Go 1.18.
- Python dependencies: `cffi Cython gmpy2 pyzmq pycryptodome pyyaml psutil reedsolo numpy pytest zfec`.
- External libraries: PBC (Pairing-Based Cryptography) library and Charm library.

A.3 Set-up

A.3.1 Installation

Setup at local machine: To locally run Dumbo-MPC, a user needs to install all dependencies listed in README.md file. Please follow the README.md file in the repository to install the necessary dependencies step by step.

Setup AWS machines: Here are the steps to set up the AWS machines for distributed tests:[1]

[1]The AWS setup is for evaluating reproducibility and can be skipped for availability and functionality evaluations.

1. Create your AWS credentials: Setup your AWS credentials to enable programmatic access to your account from your local machine. These credentials will authorize your machine to create, delete, and edit instances on your AWS account programmatically. First of all, find your access key id and secret access key. Then, create a file " /.aws/credentials" with the following content:

   ```
   [default]
   aws_access_key_id = YOUR_ACCESS_KEY_ID
   aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
   ```

2. Add your SSH publlic key to your AWS account: You must now add your SSH public key to your AWS account. This operation is manual (AWS exposes little APIs to manipulate keys) and needs to be repeated for each AWS region that you plan to use. Upon importing your key, AWS requires you to choose a "name" for your key; ensure you set the same name on all AWS regions. This SSH key will be used by the python scripts to execute commands and upload/download files to your AWS instances. If you don't have an SSH key, you can create one using ssh-keygen: `ssh-keygen -f /.ssh/aws`.

3. Deploying Dumbo-MPC on AWS: Launch an instance on AWS (with Ubuntu 20.04 LTS). If you are not familiar with AWS, you can visit Get started with Amazon EC2 Linux instances to get some help. To connect to the launched instance, use SSH:

   ```
   ssh -i your_ssh_key_path
   ubuntu@public_ip_of_instance
   ```

   Then, upload codes of Dumbo-MPC into instance:

   ```
   scp -i your_ssh_key_path -r Dumbo-MPC_code_path
   ubuntu@public_ip_of_instance:~/
   ```

   or clone Dumbo-MPC project:

   ```
   git clone https://github.com/dcy456/Dumbo-MPC.git
   ```

   Finally, you should install dependencies according to README.md, and create an image (e.g., store the image with name `your-image-id`) for this instance. If you are not familiar with AWS, you can visit Create an AMI from an Amazon EC2 Instance to get some help.

A.4 Evaluation Workflow

A.4.1 Major claims (for functionalities)

(C1): The AsyRanTriGen protocol (i.e. the pessimistic offline phase of Dumbo-MPC) is implemented, and can produce Shamir-style multiplication triples over the field of curve BLS12-381.

(C2): The entire offline phase of Dumbo-MPC is implemented, beginning with the execution of a fast path (i.e. the OptRanTriGen protocol). If the fast path fails, a fallback algorithm is triggered. Finally, AsyRanTriGen protocol (pessimistic path) will be executed to restore the robust generation of multiplication triples. All multiplication triples (generated by either OptRanTriGen or AsyRanTriGen) are of Shamir-style and over the field of curve BLS12-381.

(C3): The state-of-the-art asynchronous triple generation protocol GS23 is also implemented, and its implementation can generate multiplication triples over the field of curve secp256k1.

(C4): The multiplication triples generated by the offline phase of Dumbo-MPC (either the fast path or the pessimistic path) can be used by the online evaluations of hbMPC, such as shuffling via a butterfly network.

A.4.2 Experiments (for evaluating functionalities)

The approach creates multiple processes within a single (local) machine. Each process corresponds to one node in our protocols, and these processes communicate using an interprocess communication (ipc) channel.

(E1): A quick start to locally run AsyRanTriGen protocol to generate multiplication triples can be:

1. `cd /path/to/Dumbo-MPC`.
2. `./run_local_network_test.sh asy-triple <num_node> <batchsize>`. For this test, our artifact supports several fixed numbers of nodes, including the choices of 4, 10, 22 and 31, and arbitrary batch size. For example, running `./run_local_network_test.sh asy_triple 4 200` will generate 200 multiplication triples. The experiment logs are shown at `./dumbo-mpc/AsyRanTriGen/log/` with recording the execution latency, and the generated triples are at `./dumbo-mpc/AsyRanTriGen/triples/`. [2]

(E2): Recall that in the offline phase of Dumbo-MPC, an OptRanTriGen protocol (fast path) is firstly executed to optimistically generate multiplication triples, and if a fast-path failure occurs, a fallback happens to switch into the AsyRanTriGen protocol. So we provide the following quick start to locally run the offline protocols of Dumbo-MPC, by simulating a fast-path failure at 10-th round to trigger fallback.

1. `cd /path/to/Dumbo-MPC`.
2. `./run_local_network_test.sh dumbo-mpc <num_node> <batchsize>`. For this test, our artifact supports 4, 10, 22 and 31 nodes, and arbitrary batchsize. For example, running `./run_local_network_test.sh dumbo_mpc 4`

[2] In all subsequent experiments, the latency associated with saving the generated multiplication triples to a file is not included.

200 will generate $200*9+200=2000$ triples in total, where all nodes agree on the 9-th round of fast path and each round outputs 200 triples, and the pessimistic path outputs another 200 triples. The experiment logs are shown at ./dumbo-mpc/dualmode/log/, providing the latency for each protocol. The generated triples from both paths are stored as files under the directory ./dumbo-mpc/dualmode/opt-triples/ and ./dumbo-mpc/dualmode/asy-triples/.

(E3): A quick start to locally run GS23 protocol can be:

1. cd /path/to/Dumbo-MPC/GS23.
2. ./scripts/local_test.sh scripts/run-beaver.py <num_node> <batchsize>. For this test, our artifact supports 4, 10, 22 and 31 nodes, and batchsize $\leq$ 5000. For example, running ./scripts/local_test.sh scripts/run_beaver.py 4 200 will generate 100 multiplication triples. The experiment logs are stored at Dumbo-MPC/GS23/log/, providing the execution time.

(E4): A quick start to locally run an online task (using the online protocol inherited from hbMPC), in aid of triples generated from Dumbo-MPC's offline phase, is as follows:

1. cd Dumbo-MPC/dumbo-mpc/online.
2. ./preprocessing.sh <num_node> <num_input>. Both OptRanTriGen and AsyRanTriGen protocols are used to prepare triples and random shares, where random shares are served as inputs. For example, shuffling 64 inputs by 4 nodes requires to generate 4608 triple, which can be completed by ./proprecessing.sh 4 64. Besides, one-minus-ones shares are also required to generate for online shuffle. All preprocessed data are stored at Dumbo-MPC/dumbo-mpc/online/sharedata_test/.
3. Run ./run_shuffle.sh 4 64 for 4 nodes to shuffle 64 inputs. The experiment logs can be found at Dumbo-MPC/dumbo-mpc/online/log/.

A.5 Evaluation at Distributed AWS Instances

The above subsection outlines the approach used to evaluate the functionalities of our implementation. This subsection provides detailed guidance on reproducing the results reported in our paper. To distributedly run experiments across multiple AWS instances, first start an AWS instance (as the experiments' manager node) from the previously created AMI and modify the awsinit.sh in the directory remote/ as follows: replace -image-id with your-image-id; for -instance-type, we recommend c6a.8xlarge or better; replace -key-name with your-ssh-key-name; -security-group-ids you can delete this option if you are not familiar with it.

(AWS-E1) [Choices of batchsize] (Figure 6 and Figure 7): In this experiment, we test the relationship between throughput and batch size in the OptRanTriGen and AsyRanTriGen protocols. First, launch instances via ./awsinit.sh <num_node> where num_node can be 4, 10, 22, 31. Then, follow the next steps to evaluate OptRanTriGen and AsyRanTriGen protocols with different batch sizes.

- For OptRanTriGen, do the following steps:
  1. cd remote/OptRanTriGen_scripts.
  2. Configure ./changeconfig.sh by IP addresses of created AWS instances.
  3. Repeat to the execution of OptRanTriGen by varying <batchsize> among 1000, 5000, 10000 and 12000: ./launch_optrantrigen.sh <num_node> <batchsize>.
  4. Stop execution by ./terminate.sh, after the outputs are returned. Then, the experiment logs of all participating nodes can be downloaded by running ./scplog.sh <num_node> <batchsize>. The manger node will store the experiment logs at ./log_<num_node>_8x/test_<batchsize>/. These logs provide each node's execution time and communication cost in OptRanTriGen.
- For AsyRanTriGen, its test is similar to that of OptRanTriGen, except a couple of differences: (i) in Step 1, enter the directory AsyRanTriGen_scripts instead of OptRanTriGen_scripts; (ii) in Step 3, run the script ./launch_asyrantrigen.sh <num_node> <batchsize> instead.

(AWS-E2) [Triple throughput for LAN and WAN settings] (Figure 8 and Figure 9): This experiment evaluates the triple throughput with a fixed batch size of 5000 for four different offline protocols in both LAN and WAN settings, including OptRanTriGen and AsyRanTriGen from Dumbo-MPC, as well as GS23 and hbMPC.

First, we launch a few AWS instances via ./awsinit.sh <num_node> where num_node can be 4, 10, 22, 31. The testing processes in the LAN and WAN settings are almost same, except that in the WAN setting, we first run the bash script set_latency.sh in the remote directory, which uses Linux TC tool to restrict the upload bandwidth of each instance to 500 Mbps and set the network's RTT to 150 ms.

Then we evaluate four offline protocols with a fixed batch size of 5000 as follows.

1. cd remote/<AsyRanTriGen_scripts>, where the directory of scripts <OptRanTriGen_scripts> can be one of GS23_scripts, GS23_scripts, GS23_scripts, hbMPC_scripts.
2. Configure ./changeconfig.sh by IP addresses of created AWS instances.

3. Launch the specific offline protocol: `./<run_script> <num_node> 5000`. Here the parameter `<run_script>` is a bash script from `launch_asyrantrigen.sh`, `launch_optrantrigen.sh`, `launch_GS23.sh`, and `launch_hbmpc.sh`, representing the scripts to launch OptRanTriGen, AsyRanTriGen, GS23 and hbMPC, respectively.

4. Stop running by `./terminate.sh`, after the execution finishes. Then, we can read the experiment results by connecting these instances to view these logs or running `./scplog.sh <num_node> 5000` to download these logs. These logs will be stored under `./log_<num_node>_8x/test_5000/` at the manger node. The experiment logs provide the execution time and communication costs of each participating node.

(AWS-E3) [Communication cost] (Figure 10): In AWS-E2, we already evaluated the performance of four offline protocols. Their communication cost can be found in the test logs.

(AWS-E4) [Performance with fallback (Figure 11)]: First, launch AWS instances via the script `./awsinit.sh <num_node>` where `num_node` is set as 4. Then, evaluate three protocols (Dumbo-MPC, GS23 and hbMPC) with a fixed batch size of 5000 for `num_node`=4.

- Run Dumbo-MPC's offline phase (i.e. OptRanTriGen with fallback to AsyRanTriGen): The experiment design is similar to that of E-2. First, the fast-path protocol (OptRanTriGen) is executed to optimistically generate multiplication triples, until a malicious node is simulated at the 10-th round of OptRanTriGen. Following this, all nodes switch to the pessimistic path through a secure fallback mechanism, ensuring agreement on the triples generated during the fast path. Finally, the AsyRanTriGen protocol (pessimistic path) is executed by all nodes.

  1. `cd Dumbo-MPC_scripts`
  2. Configure `./changeconfig.sh` by IP addresses of created AWS instances.
  3. Launch Dumbo-MPC: `./launch_dumboMPC.sh 4 5000`.
  4. Terminate Dumbo-MPC and access to the experiment results: Run `./terminate.sh` to terminate all instances. Then, we can run the script `./scplog.sh 4 5000` to download the experiment logs from all participating nodes. These logs will be stored at `./log_4_8x/test_5000/` at the manger node. The logs provide the execution time for each step, including every round of the OptRanTriGen protocol, the fallback process, and the AsyRanTriGen protocol. Besides, the communication costs of OptRanTriGen and AsyRanTriGen are separately enumerated, and can be found at logs.

- Run GS23: same as running GS23 for the LAN setting in AWS-E2.

- Run hbMPC: same as running hbMPC for the LAN setting in AWS-E2.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: FABLE: Batched Evaluation on Confidential Lookup Tables in 2PC

Zhengyuan Su*, Qi Pang†, Simon Beyzerov†, Wenting Zheng†
*Tsinghua University, †Carnegie Mellon University

A Artifact Appendix

A.1 Abstract

This artifact provides the code and instructions to reproduce the major claims of FABLE, an efficient 2PC protocol for confidential LUT evaluation, with lightweight client computation, concretely efficient server computation, scalable communication, and LUT confidentiality. In particular, it provides the instructions and scripts to reproduce all evaluation results of FABLE and its baselines. It also contains easy-to-use scripts to reproduce FABLE's advantages in the two applications mentioned in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

To the best of our knowledge, executing this artifact does not impose any risk on the machine's security, data privacy, or any other ethical concern. The artifact showcases an efficient 2PC protocol that helps enhance user privacy in real-world collaborative computation. We believe the artifact is safe to execute.

A.2.2 How to access

The artifact is available at https://doi.org/10.5281/zenodo.15586635, which contains two git repositories.

If you prefer to download from GitHub, please clone the repositories https://github.com/cmu-cryptosystems/FABLE and https://github.com/cmu-cryptosystems/FABLE-AE, and place them side by side under the same folder.

A.2.3 Hardware dependencies

Our code does not require specific hardware features to compile and run. However, to help reproduce the numbers shown on the paper, we provide *two c6i.16xlarge instances on AWS EC2*.

We would like to provide the reviewers with access via SSH to our two `c6i.16xlarge` instances, which contain the environments to execute the code. To gain access to the machines, please follow the instructions on the HotCRP system.

A.2.4 Software dependencies

In our artifact, we provide multiple Dockerfiles to set up the environments.

To set up the environment to build and run FABLE, build the Dockerfiles under the two folders:

```
docker build ./FABLE -t fable:1.0
docker build ./FABLE-AE -t fable-ae:1.0
```

Note that you may need to run the commands under sudo if you are not a root user.

Some baselines, including the DORAM baselines and Crypten's application baseline, require separate environments to execute. However, they are not mandatory because we provide their performance numbers in the artifact. If you prefer to reproduce those numbers, please run the following commands to build the Docker images:

```
bash FABLE-AE/ORAMs/build-dockers.sh
docker build FABLE-AE/Crypten -t crypten
```

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

FABLE is installed inside the Docker environment while the image `fable:1.0` is built. Therefore, no operation is needed for installation other than building the Dockerfile.

A.3.2 Basic Test

To check that the installation is successful, first start a Docker container with

```
docker run -it -w /workspace/FABLE fable:1.0
```

Inside the container, run the following commands:

```
./build/bin/fable 127.0.0.1 r=1 l=1 &
./build/bin/fable 127.0.0.1 r=2 l=1
```

It will run a toy example of FABLE in 10 seconds, report the breakdown of computation and communication, and check the result.

A.4 Evaluation workflow

A.4.1 Major Claims

The major claims of FABLE consist of the following:

(C1): FABLE achieves orders of magnitude speedup over prior baselines under both LAN and WAN networks. In particular, FABLE achieves up to 28-101× speedup in LAN environments and up to 50-393× speedup in WAN environments. This is proven by the end-to-end performance evaluation described in Section 6.3 and Figure 5.

(C2): FABLE's performance benefits from multi-threading and batched processing. The benefit from multi-threading is described in Section 6.2 and Figure 4. The benefit from batched processing is described in Section 6.3, Figure 6, and Figure 7.

(C3): FABLE's performance remains competitive even when the OPRF is replaced with the more secure AES, as shown in Table 4.

(C4): FABLE also exhibits orders of magnitude speedup under two applications. As explained in Section 6.4, FABLE achieves a speedup of 456× (178×) for secure embedding lookup and a speedup of 15× (27×) for secure query execution under LAN (WAN).

A.4.2 Experiments

(E1): [Main Experiments] [30 human-minutes + 10 compute-hour + 1GB disk]: run FABLE under various configurations and the SPLUT baseline, and reproduce the main plots.

Preparation: In this experiment, we use the `fable-ae` image. Specifically, in the root directory of our artifact, launch a container with

```
docker run -it --net=host --cap-add=NET_ADMIN
    -v $PWD/FABLE-AE:/workspace/AE -w
    /workspace/AE fable-ae:1.0
```

on two instances. Then, set two environment variables on both instances: `HOST` and `CLIENT`, which are the IP addresses of the server and the client, respectively.

Execution: To launch the main experiments, run

```
bash main_experiments.sh 1 0.0.0.0
```

and

```
bash main_experiments.sh 2 $HOST
```

on the server and the client, respectively.
After the main experiments finish, please launch the SPLUT experiments with

```
bash baseline_splut.sh 1 0.0.0.0
```

and

```
bash baseline_splut.sh 2 $HOST
```

on the server and the client, respectively.
Then, synchronize the experiment results with

```
bash sync_log_recv.sh $CLIENT
```

and

```
bash sync_log_serv.sh
```

on the server and the client, respectively.
Finally, produce the plots and the tables with

```
python3 microbench.py
python3 draw.py
python3 breakdown_table.py
python3 breakdown_table.py --aes
```

on the server.

Results: You should find 60 log files under `FABLE-AE/logs/main` and 12 log files under `FABLE-AE/logs/baseline/splut` on both instances. On the server side, you will also find the following plots under `FABLE-AE/plots`:
Figure 4: `runtime_vs_threads.pdf`.
Figure 5: `time_lutsize_network_clipped.pdf`.
Figure 6: `24_time_bs.pdf`.
Figure 7: `comm.pdf`.
Also, the execution of `breakdown_table.py` prints Table 4 in the standard output.

(E2): [Application] [30 human-minutes + 1.5 compute-hour]: run the applications and reproduce the speedups.

Preparation: In this experiment, we still use the `fable-ae` image. Specifically, in the root directory of our artifact, launch a container with

```
docker run -it --net=host --cap-add=NET_ADMIN
    -v $PWD/FABLE-AE:/workspace/AE -w
    /workspace/AE fable-ae:1.0
```

on two instances. Then, set two environment variables on both instances: `HOST` and `CLIENT`, which are the IP addresses of the server and the client, respectively.

Execution: To launch the application experiments, run

```
bash applications.sh 1 0.0.0.0
```

and

```
bash applications.sh 2 $HOST
```

on the server and the client, respectively.
Afterwards, run `python3 read_app_speedup.py` to see the speedup for both applications.

Results: You should find 6 log files under `FABLE-AE/logs/applications` on both instances.

The Python file will print the speedups of both applications in the standard output, which should be over 100× for the embedding lookup and over 10× for the secure join execution.

(E3): [Optional Reproduction of DORAM Baselines] [30 human-minutes + 1.5 compute-hour]: reproduce the DORAM baseline performance. This step is *optional* as we already provide the evaluation results in our artifact.
Preparation: In this experiment, we use DUORAM's image. Please ensure that you have already executed

```
bash FABLE-AE/ORAMs/build-dockers.sh
```

Execution: To launch the DORAM baseline experiments, run

```
bash FABLE-AE/ORAMs/repro-dockers.sh
```

on the server.
Afterwards, run `python3 draw.py --doram-baseline` to see the plots with the DORAM baseline logs.
Results: You should find 12 log files under `FABLE-AE/logs/baseline/floram` and another 12 log files under `FABLE-AE/logs/baseline/duoram`. You will find the same set of plots as E1.

(E4): [Optional Reproduction of the Crypten baselines] [30 human-minutes + 1.5 compute-hour]: reproduce the Crypten baseline performance. This step is also *optional* as we already provide the evaluation results in our artifact.
Preparation: In this experiment, we use the `crypten` image. Please launch the Docker container on two instances:

```
docker run -it --net=host --cap-add=NET_ADMIN
    -v $PWD/FABLE-AE:/workspace/AE -w
    /workspace/AE crypten
```

Then, set two environment variables on both instances: `HOST` and `CLIENT`, which are the IP addresses of the server and the client, respectively.
Execution: To launch the experiments, run

```
bash Crypten/reproduce.sh 0 $HOST $CLIENT
```

and

```
bash Crypten/reproduce.sh 1 $HOST $CLIENT
```

on the server and the client, respectively.
Afterwards, run

```
python3 read_app_speedup.py --crypten-baseline
```

to see the speedup for both applications.
Results: You should find 10 log files in total under `FABLE-AE/logs/applications`. The speedups will be similar to E2.

A.5 Notes on Reusability

The reproduction only uses a subset of all flags supported in FABLE. In this section, we elaborate the flags of the executable `build/bin/fable`.

- `p`: the port number for communication. Default: 8000.
- `bs`: the batch size of the input. Default: 4096.
- `db`: the LUT size. Default: `2^LUT_INPUT_SIZE`, i.e., the maximum supported size.
- `seed`: the random seed. Default: 12345.
- `par`: Whether enable parallelization. Default: 1 (i.e., enable parallelization).
- `thr`: Number of threads used in parallelization. Only useful when `par=1`, Default: 16.
- `l`: Type of the LUT. It will not affect the performance of FABLE. See the README for more information. Default: 0 (i.e., Random LUT).
- `h`: The OPRF type. Default: 0 (i.e., LowMC).
- `f`: Whether to do operator fusion to save communication rounds. After operator fusion, we cannot observe the detailed breakdown performance, as the operations are fused together. Default: 0 (i.e., no fusion).

In addition, one can also reuse 2PC subprotocols of FABLE (e.g., deduplication and OPRF evaluation) as building blocks for other protocols. They are implemented under `FABLE/src/GC` and built upon EzPC.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: MAESTRO: Multi-Party AES Using Lookup Tables

Hiraku Morita
Aarhus University
University of Copenhagen

Erik Pohle
COSIC, KU Leuven

Kunihiko Sadakane
The University of Tokyo

Peter Scholl
Aarhus University

Kazunari Tozawa
The University of Tokyo

Daniel Tschudi
Concordium
Eastern Switzerland University of Applied Sciences (OST)

A Artifact Appendix

A.1 Abstract

Secure multi-party computation (MPC) enables multiple distrusting parties to jointly compute a function while keeping their inputs private. Computing the AES block cipher in MPC, where the key and/or the input are secret-shared among the parties is important for various applications, particularly threshold cryptography.

In this work, we propose a family of dedicated, high-performance MPC protocols to compute the non-linear S-box part of AES in the honest majority setting. Our protocols come in both semi-honest and maliciously secure variants. Our implementation of the MPC protocols for three parties will be the main artifact.

Our protocols have different trade-offs, such as having a similar round complexity as previous state-of-the-art by Chida et al. [WAHC'18] but 37% lower bandwidth costs, or having 27% fewer rounds and 16% lower bandwidth costs. An experimental evaluation in various network conditions using three party replicated secret sharing shows improvements in throughput between 28% and 71% in the semi-honest setting. For malicious security, we improve throughput by 319% to 384% in LAN and by 717% in WAN due to sublinear batch verification.

A.2 Description & Requirements

The artifact is software, given as source code, and the raw benchmark data. The software part implements all protocols that were evaluated experimentally and the raw benchmark data is the raw resulting runtime/communication cost, etc. when running the software on our servers.

A.2.1 Security, privacy, and ethical concerns

The source code provided does not pose any risk. There is a minimal risk of a supply chain attack as the code uses external libraries. We are not aware of any privacy or ethical concerns arising from benchmarking or testing our code.

A.2.2 How to access

The source code and raw benchmark data can be found in https://github.com/KULeuven-COSIC/maestro, or archived in https://doi.org/10.5281/zenodo.14719154. The `README` file in the root directory provides instructions on how to use the artifact and how to parse the raw benchmark data.

A.2.3 Hardware dependencies

Evaluation of this artifact does not require special hardware. Benchmarking requires three servers/computers with `x86` or `AArch64` architecture offering CLMUL support (see Section A.3.2 on how to test for this). The throughput and latency of the network connecting the servers are specific to the different benchmarks. The settings range from "WAN network" (50 Mbits/s and 100 ms round-trip time) to "LAN network" (10 Gbit/s and almost no latency). More details are found in the `README` file.

The benchmark results in the paper were created on three separate servers with 16 CPU cores and 128GB RAM. To validate our claims, in particular (C1) for functionality, it is not necessary to have access to similar hardware. To validate (C2) or (C3), three machines with 4 to 8 cores, and 8GB to 16GB RAM should suffice to run the protocols with smaller batch sizes.

A.2.4 Software dependencies

Our software was developed on Linux and MacOS using the Rust programming language. All dependencies are freely available online and are described in the `README` file.

A.2.5 Benchmarks

The raw data from the experiments reported in the paper can be found in the `benchmark-data` folder. We consider throughput and latency benchmarks. The details of the data format are described in the `README` file mention above.

Throughput Benchmarks

- `benchmark-data/10Gbit` contains data of all protocols in the 10 Gbit/s network with batch sizes 50000, 100000 and 250000.
- `benchmark-data/1Gbit` contains data of all protocols in the 1 Gbit/s network with batch sizes 50000, 100000 and 250000.
- `benchmark-data/200Mbps-15msRTT` contains data of all protocols in the 200 Mbit/s with 15 ms round trip time network with batch sizes 50000, 100000 and 150000.
- `benchmark-data/100Mbps-30msRTT` contains data of all protocols in the 100 Mbit/s with 30 ms round trip time network with batch sizes 10000, 50000 and 100000.
- `benchmark-data/50Mbps-100msrtt` contains data of all protocols in the WAN network (50 Mbit/s with 100 ms round trip time) with batch sizes 10000, 50000 and 100000.

Latency Benchmarks

- `benchmark-data/10Gbit-latency` contains data for 1 AES block in the 10 Gbit/s network,
- `benchmark-data/1Gbit-latency` contains data for 1 AES block in the 1 Gbit/s network,
- `benchmark-data/200Mbps-15msRTT-latency` contains data for 1 AES block in the 200 Mbit/s with 15 ms round trip time,
- `benchmark-data/100Mbps-30msRTT-latency` contains data for 1 AES block in the 100 Mbit/s with 30 ms round trip time,
- `benchmark-data/50Mbps-100msrtt-latency` contains data for 1 AES block in the WAN network.

A.3 Set-up

All information on installation, tests, and evaluation workflow are also found in the `README` file in the root directory of the artifact.

A.3.1 Installation

1. Download the artifact from the repository provided above.
2. Install a recent version of OpenSSL.
3. Install Rust, version 1.75 or newer.
4. Install a recent version of Python 3 and the `numpy`, `pandas` packages (to parse benchmark results).

A.3.2 Basic Tests

1. Use `RUSTFLAGS='-C target-cpu=native' cargo test --lib` to run basic unit tests.
2. Use `RUSTFLAGS='-C target-cpu=native' cargo bench "CLMUL Multiplication"` to check that your machine offers hardware support for carry-less multiplication. If the test succeeds your machine has CLMUL support.

A.4 Evaluation workflow

To prepare for benchmarking proceed as follows:

1. Compile the benchmark binary using `RUSTFLAGS='-C target-cpu=native' cargo build -release --bin maestro --features="clmul"` (this assumes that the machine compiling and the benchmark servers are using the same architecture).
2. On each benchmark server setup the necessary TLS certificates.
3. If necessary, modify the network conditions.
   (a) Run `ip addr show` (or a similar command) to obtain the name of the network interface of each benchmark server, e.g., `eth0`, `enp0s3`, etc.
   (b) In case a previous experiment modified the network, remove the limitation before adding a new one: `tc qdisc del dev <iface name> root netem`
   (c) Set bandwidth and round trip time (RTT) as `tc qdisc add dev <iface name> root netem rate <bandwidth> delay <0.5 * RTT>`, for example `tc qdisc add dev eth0 root netem rate 200mbit delay 7.5ms` to limit the network traffic over eth0 to 200Mbit/s with a round trip time of 15ms.

4. Start the benchmark via CLI on all three server.

More details are given in the `README` file mentioned above.

A.4.1 Major Claims

(C1): *The artifact is functional. The provided source code compiles successfully and the unit tests run without errors. Using three terminals, one can run any of the implemented protocols on the same machine where communication is over localhost. This is proven by experiment (E1)*

(C2): *The throughput benchmarks are reproducible. That is one can replicate the relative throughput of the different protocols, i.e., claims of the form "Protocol X has twice the throughput than Protocol y". This is proven by experiment (E2).*

(C3): *The latency benchmarks are reproducible. That is one can replicate the relative latency of the different protocols. This is also proven by experiment (E2).*

A.4.2 Experiments

(E1): *[Run a small-scale benchmark: 20 human minutes + 10 compute minutes]* Experiment E1 builds upon the successful compilation of the source code and execution of unit tests (from Section A.3.2).
Set B to be the batch size, say 1000. Set R to be the number of repetitions, say 2.

1. Follow the description *Running benchmarks > On localhost* in the `README` file.
2. Open three terminals on the evaluation machine. Then run the benchmark binary with SIMD B and rep R, like this `target/release/maestro --config p1.toml --threads 4 --simd B --rep R --csv result-p1.csv chida lut16 gf4-circuit lut256 lut256-ss mal-chida mal-chida-rec-check mal-lut16-ohv mal-gf4-circuit-opt mal-lut256-ss-opt` (repeat for the second and third terminal with `p2.toml` and `p3.tomls`, and with `result-p2.csv` and `result-p3.csv`, respectively.
3. There should be information printed in all terminals about which protocol is currently benchmarked, etc.
4. When all finished successfully, it should say `Writing CSV-formatted benchmark results to result-p1.csv`.
5. Parse the benchmark data using `python parse-csv.py result-p1.csv result-p2.csv result-p3.csv`.

If all these steps ran without error and the parsing script shows throughput for the different protocols, this experiment is successful.

(E2): *[Run the Benchmarks] [2 human hours + 5 compute hour]: Experiment E2 is to run the benchmarking software given the different network conditions mention in Section A.2.5.*
Computing on larger batch sizes takes more time, in particular in networks with latency. Thus for (E2) we propose reduced parameter numbers and repetitions for each scenario.

- (Throughput) 10Gbit network: $B = 100000$ with $R = 5$.
- (Latency) 10Gbit network: $B = 1$ with $R = 10$.
- (Throughput) 1Gbit network: $B = 50000$ with $R = 5$.
- (Latency) 1Gbit network: $B = 1$ with $R = 10$.
- (Throughput) 200Mbit network: $B = 10000$ with $R = 5$.
- (Latency) 200Mbit network: $B = 1$ with $R = 10$.
- (Throughput) 100Mbit network: $B = 10000$ with $R = 5$.
- (Latency) 100Mbit network: $B = 1$ with $R = 10$.
- (Throughput) 50Mbit network: $B = 10000$ with $R = 5$.
- (Latency) 50Mbit network: $B = 1$ with $R = 10$.

Preparation: Setup the network and servers and prepare the benchmark software using the steps outlined in Section A.3.

Execution:

1. Setup the latency and throughput of the network.
2. Run the benchmark software on all three machines.
3. Repeat for each of the claimed network settings.

Results:

1. Parse the benchmark results using the given python script.
2. Compare the outcome with the original data in the `benchmark-data` folder. The relative differences in throughput and latency of the different protocols should roughly match.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Efficient Multi-Party Private Set Union Without Non-Collusion Assumptions

Minglang Dong[1,2,3], Cong Zhang[4], Yujie Bai[1,2,3], and Yu Chen[1,2,3](✉)

[1]School of Cyber Science and Technology, Shandong University, Qingdao 266237, China
[2]Quan Cheng Laboratory, Jinan 250103, China
[3]Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao 266237, China
[4]Institute for Advanced Study, BNRist, Tsinghua University, Beijing, China
{minglang_dong,baiyujie}@mail.sdu.edu.cn, zhangcong@mail.tsinghua.edu.cn, yuchen@sdu.edu.cn

A Artifact Appendix

A.1 Abstract

We present an open-source C++ implementation of the two MPSU protocols proposed in our paper. The artifact includes the complete source code and configurations needed to replicate our experiments in Section 7. We also provide detailed setup instructions for installing dependencies, building the project, and running the protocols. Our implementations enable evaluators to run these two protocols with customizable parameters, such as number of parties and set size, to evaluate the online and offline performance under various settings. The artifact is available, functional, and reproducible. We welcome suggestions and performance reports for future reproducibility.

A.2 Description & Requirements

Our artifact is structured as follows:

- SKMPSU/ and PKMPSU/: Complete implementations of our SK-MPSU and PK-MPSU protocols, respectively.
- README.md: Step-by-step guides for dependency installation, project compilation, and execution.
- autoTestshell/: Automated scripts to test both SK-MPSU and PK-MPSU protocols, allowing for quick and efficient evaluation across different configurations.

Our implementations support up to 10 parties with sets of up to 2^{20} items each, on machines with 128GB memory. This has been tested on Ubuntu 22.04 with g++ 11.4.0.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Our implementations are open-source on GitHub at https://github.com/real-world-cryptography/MPSU. Our artifact is also available at https://doi.org/10.5281/zenodo.14694832.

A.2.3 Hardware dependencies

All experiment results of our paper are obtained by running the artifact on a single server with 32-cores Intel Xeon 2.70GHz CPU and 128GB of RAM.

A.2.4 Software dependencies

Our implementations are built on Vole-PSI (https://github.com/Visa-Research/volepsi) and require additional library dependencies including OpenSSL (https://github.com/openssl/openssl) and OpenMP (https://www.openmp.org) libraries. They have been tested on Ubuntu 22.04 with installations of g++ 11.4.0 and CMake 3.22.1.

A.2.5 Benchmarks

We set the computational security parameter $\lambda = 128$ and the statistical security parameter $\sigma = 40$. We test the balanced scenario by setting all input sets to be of equal size. In our SK-MPSU, the element are 64-bit strings. In our PK-MPSU, the elements are encoded as EC points in compressed form.

A.3 Set-up

A.3.1 Installation

The complete installation guide with all dependencies and build instructions is available in our GitHub repository: https://github.com/real-world-cryptography/MPSU/blob/main/README.md

A.3.2 Basic Test

A.3.3 Test Parameters

We use the following notations for test parameters:

- `nn` $\in \{10,12,14,16,18,20\}$: Logarithm of set size $\{2^{10},2^{12},2^{14},2^{16},2^{18},2^{20}\}$.
- `nt` $\in \{1,2,4,8\}$: Number of threads
- `k`: Number of parties
- `r` $\in \{0,\ldots,$ `k` $-1\}$: Party index

A.3.4 SK-MPSU Test.

To test SK-MPSU for 3 parties with sets of 2^{12} items each in a single thread, execute the following commands in the `SKMPSU/build` directory:

- `./main -genSC -k 3 -nn 12`
- `./main -preGen -k 3 -nn 12 -r 0 & ./main -preGen -k 3 -nn 12 -r 1 & ./main -preGen -k 3 -nn 12 -r 2`
- `./main -psu -k 3 -nn 12 -r 0 & ./main -psu -k 3 -nn 12 -r 1 & ./main -psu -k 3 -nn 12 -r 2`

A.3.5 PK-MPSU Test.

To test PK-MPSU for 3 parties with sets of 2^{12} items each in a single thread, execute the following commands in the `PKMPSU/build` directory:

- `./main -preGen -k 3 -nn 12 -r 0 & ./main -preGen -k 3 -nn 12 -r 1 & ./main -preGen -k 3 -nn 12 -r 2`
- `./main -psu -k 3 -nn 12 -r 0 & ./main -psu -k 3 -nn 12 -r 1 & ./main -psu -k 3 -nn 12 -r 2`

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): In our paper, we claimed that we fully re-implement the state-of-the-art MPSU protocols in C++. This claim can be verified by running the experiments with different configurations in Section A.4.2.

(C2): Before executing the MPSU program, the required offline files must be generated correctly. Therefore, it is crucial to run the commands in the specified sequence.

A.4.2 Experiments

(E1): *[Config network settings] Config network settings using tc.*
How to: Open a terminal, and execute the following command: `tc qdisc add dev lo root netem delay 80ms rate 400Mbit`. Then, the local network is configured as 400Mbit bandwidth with 80ms latency. Evaluators can try other network settings with other parameters, e.g., 50Mbit/80ms, 1Gbit/40ms, 10Gbit/0.02ms.
Results: Execute "sudo tc qdisc show dev lo" to see if the network is configured correctly.

(E2): *[Run SK-MPSU protocol] : Benchmark the running time and communication cost of SK-MPSU protocol across diverse scenarios.*
How to: To test SK-MPSU for 3 parties with sets of 2^{12} items each in a single thread, execute the following commands in the `SKMPSU/build` directory:

- `./main -genSC -k 3 -nn 12`
- `./main -preGen -k 3 -nn 12 -r 0 &`
  `./main -preGen -k 3 -nn 12 -r 1 &`
  `./main -preGen -k 3 -nn 12 -r 2`
- `./main -psu -k 3 -nn 12 -r 0 & ./main -psu -k 3 -nn 12 -r 1 & ./main -psu -k 3 -nn 12 -r 2`

Results: The output messages during execution are as follows:

- `generate sc done`. This indicates the Share Correlation files have been generated successfully.
- `P1 offline preGen time cost = 13.142 s`. This indicates the running time of generating offline files (including Vole Triples, GMW Triples and ROT) is 13.142 s.
- `P1 offline preGen communication cost = 1.829 MB`. This indicates the communication cost of generating offline files (including Vole Triples, GMW Triples and ROT) is 1.829 MB.
- `P1 communication cost = 1.690 MB`. This indicates the online communication cost of the leader in SK-MPSU protocol is 1.690 MB.
- `end  3157.4  3157.364 **********` . This indicates the online running time of the leader in SK-MPSU protocol is 3157.4 ms.
- `Success! union size: 4098`. This indicates the SK-MPSU protocol successfully completed.

(E3): *[Run PK-MPSU protocol] : Benchmark the running time and communication cost of PK-MPSU protocol across diverse scenarios.*
How to: To test PK-MPSU for 3 parties with sets of 2^{12} items each in a single thread, execute the following commands in the `PKMPSU/build` directory:

- `./main -preGen -k 3 -nn 12 -r 0 &`
  `./main -preGen -k 3 -nn 12 -r 1 &`
  `./main -preGen -k 3 -nn 12 -r 2`
- `./main -psu -k 3 -nn 12 -r 0 & ./main -psu -k 3 -nn 12 -r 1 & ./main -psu -k 3 -nn 12 -r 2`

Results: The output messages during execution are as follows:

- `P1 offline pre time cost = 0.984 s.` This indicates the running time of generating offline files (including GMW Triples) is 0.984 s.
- `P1 offline pre communication cost = 0.334 MB`. This indicates the communication cost of generating offline files (including GMW Triples) is 0.334 MB.
- `P1 communication cost = 4.956 MB`. This indicates the online communication cost of the leader in PK-MPSU protocol is 4.956 MB.
- `end 7984.2 7983.944 **********`. This indicates the online running time of the leader in PK-MPSU protocol is 7984.2 ms.
- `Success! union size: 4098`. This indicates the PK-MPSU protocol successfully completed.

A.5 Notes on Experimental Results

Our latest local tests demonstrate a minor improvement in the online performance (computation and communication) of our implementations compared to the results reported in the paper. This difference stems from an update to Vole-PSI.

We retained the original reported data in the paper primarily because we became aware of this update after the camera-ready submission deadline. Additionally, the performance improvement is minor, and future updates to Vole-PSI or other library dependencies may continue to introduce slight variations in results compared to those in the paper.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Scalable Collaborative zk-SNARK and Its Application to Fully Distributed Proof Delegation

Xuanming Liu[1], Zhelei Zhou[1], Yinghao Wang[1], Yanxin Pang[3], Jinye He[4],
Bingsheng Zhang[1], Xiaohu Yang[1], Jiaheng Zhang[2]
[1]*Zhejiang University* [2]*National University of Singapore* [3]*Tsinghua University* [4]*University of Virginia*
{hinsliu, zl_zhou, asternight, bingsheng, yangxh}@zju.edu.cn,
jhzhang@nus.edu.sg, pangyx21@mails.tsinghua.edu.cn, qfn5bh@virginia.edu

A Artifact Appendix

A.1 Abstract

In a nutshell, this work enables many low-resource servers to jointly execute a *multiparty computation* (MPC) protocol for zk-SNARK proof generation. During the proof generation, the servers collaboratively compute a zk-SNARK proof for a large circuit while keeping their secret inputs—known as *the witness* in zk-SNARK terminology—hidden from each other. A key feature of our design is its *scalability*: each server incurs less computational and memory overhead compared to the original monolithic prover.

This artifact provides a proof-of-concept implementation. It includes a Rust-based prototype, supporting collaborative proof generation. The artifact includes implementations of collaborative primitives, the packed secret sharing scheme, and the complete collaborative zk-SNARK (HyperPlonk), which are introduced in the paper. Moreover, it provides two modes of execution: (i) Local mode and (ii) Distributed mode, allowing users to simulate the protocol either on a single machine or in a distributed network across multiple machines. Refer to the `README.md` file for detailed instructions on how to run the artifact. It also includes scripts to benchmark performance under various deployment configurations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact is an academic prototype and is not suitable for production uses. It has not undergone formal reviews.

Although it involves cryptographic protocols and peer-to-peer communication, it does not perform any destructive or malicious operations on developers' machines. However, since the artifact requires network-based interactions between multiple parties, we recommend running it in a controlled environment, virtual machine or sandbox to mitigate potential risks related to system security or data privacy during testing.

A.2.2 How to access

The artifact is maintained at GitHub. A branch named `artifact-eval`, containing the code and docs specifically prepared for AE, is available at: https://github.com/LBruyne/Scalable-Collaborative-zkSNARK/tree/artifact-eval.

For artifact evaluation, we also provide a stable reference to the evaluated version. The direct link to this version is available on Zenodo: https://doi.org/10.5281/zenodo.16722573. We recommend reviewers download the artifact from the Zenodo link, as it includes additional documentation and detailed instructions for usage.

A.2.3 Hardware dependencies

This artifact does not require special hardware features.

As detailed in the `README.md` file, the artifact supports two modes of execution. Reviewer/User could choose the mode that best fits their testing environment and available resources.

- **Distributed execution mode (Benchmark):** This mode really runs a distributed network and supports large-scale deployment across 16-128 virtual or physical machines to run the collaborative proof generation. Each peer can be a low-resource instance with only 4GB of memory. All peers must be connected over a LAN or WAN network.
- **Local execution mode (Local/Leader):** (Recommended for its simplicity) Suitable for small-scale testing, this mode runs on a single machine. It does not require any special hardware beyond a standard system with sufficient computational resources. For reproducibility concerns, we recommend a machine with at least 1TB of RAM.

A.2.4 Software dependencies

This artifact is developed using the Rust nightly toolchain. The following specific versions were used for development and evaluation (but other versions may also work):

- `rustup` 1.27.1 (54dd3d00f 2024-04-24)

- `cargo` 1.80.0-nightly (05364cb2f 2024-05-03)

For convenience, we recommend installing the utility tool `just`, which simplifies execution and scripting. It can be installed from https://github.com/casey/just.

A Linux-based operating system is recommended. All development and testing were conducted on Ubuntu 22.04 LTS.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

First, download the artifact and install the required Rust nightly toolchain. We also recommend installing the `just` tool to simplify command execution.

Then, set up the environment with the following command:

```
rustup default nightly-2024-05-03 # You can use a
    different nightly version, but this is the one
    we used for development.
```

Next, build the artifact using:

```
cargo build --release
```

This command compiles the Rust code and prepares the executable files for use.

Finally, run a basic test to ensure the artifact is set up correctly:

```
just run --release --example sumcheck -F leader -- --
    l 4 --n 10
```

If you do not have `just` installed, you can run the following command instead:

```
RUSTFLAGS="-Ctarget-cpu=native -Awarnings" cargo +
    nightly run --release --example <example_name> <
    args>
```

where `<example_name>` is the name of the example (e.g., `sumcheck`), and `<args>` are the relevant command-line arguments (e.g., `-F leader - -l 4 -n 10`).

You may need to use the following commands to allow the user's shell to run the artifact properly:

```
ulimit -HSn 65536
```

This command runs a simple example of the collaborative sumcheck with input size 2^{10} in **leader** mode (refer to the `README.md` file), verifying that the main components of the artifact are functioning correctly. The output looks like this:

```
Start: Local Sumcheck (thread ThreadId(1))
End: Local Sumcheck (thread ThreadId(1)) ...s
Start: Collaborative Sumcheck Leader (thread
    ThreadId(1))
...
End: Collaborative SumcheckProduct Leader (
    thread ThreadId(1)) ...s
Comm: (...)
```

A.3.2 Basic Test

To test the artifact's functionality, run:

```
just run --release --example hyperplonk -F local --
    --l 8 --n 16
```

This command executes a moderate-size example of the collaborative HyperPlonk protocol in **local** mode (refer to the `README.md` file), verifying that the artifact is functioning correctly. This simulates the workload of $64 = 8 \times l$ parties on a $2^{16} = 2^n$-gate circuit *one-party-by-one-party* using a *single thread*. The full network is emulated locally.

The expected output should resemble:

```
Start: Local HyperPlonk (thread ThreadId(1))
...
End: Local HyperPlonk (thread ThreadId(1)) ...s
Start: Local HyperPlonk++ (thread ThreadId(1))
...
End: Local HyperPlonk++ (thread ThreadId(1)) ...
    s
Start: Simulate Collaborative Hyperplonk++ (
    thread ThreadId(1))
...
Comm: (...)
Comm: (...)
End: Simulate Collaborative Hyperplonk++ (thread
     ThreadId(1)) ...s
```

This command typically takes around 8 minutes to complete. You can notice that according to the log file, there are actually three instances being run: local HyperPlonk, local HyperPlonk++, and collaborative HyperPlonk++. The runtime duration is the runtime sum of these three instances. If it does not run successfully, please refer to the `README.md` file for troubleshooting. A common issue is insufficient memory allocation for virtual machines—try a smaller value of n, or use a machine with more memory.

A.4 Evaluation workflow

A.4.1 Major Claims

The protocol enables scalable collaborative zk-SNARK:

(C1): Each peer in the collaborative zk-SNARK (in this paper, HyperPlonk) incurs only a fraction ($O(\frac{1}{N})$, where N is the number of servers) of the time and memory overhead compared to the monolithic prover. The communication is small. This is demonstrated in experiment (E1), corresponding to Section 5.2, 6.2, Figure 3, and Table 2, 3 of the paper.

A.4.2 Experiments

First, we introduce (this can also be found in the `README.md` file) that the artifact provides two modes of execution:

- **Distributed execution mode:** The Rust feature **Benchmark**, enabled by `-F benchmark` in the command, specifies the protocol is run in a real distributed network. When benchmarking using this mode (this is what we do in the experiments), you need sufficient servers or machines connected over a LAN/WAN network. In the `README.md` file, we provide very detailed scripts to set up such a network and run the benchmarks. Especially note that there are many files and paths needed to by renamed, which are listed in the `README.md` file.
- **Local execution mode:** However, we understand reviewers may not have access to a distributed network or sufficient machines. Therefore, we also provide a local execution mode, which simulates the protocol in a single machine. The Rust feature **Local**, specifies the protocol is simulated locally. In this mode, A single thread will run each peer's workload one-by-one in circulation. This mode does not require a network connection.

Next, we describe the recommended experiments:

(E1): [Collaborative vs. Monolithic Prover] [30 human-minutes + 24 compute-hours]: This experiment compares the performance of collaborative HyperPlonk against the monolithic prover on general circuits.

If you are running the benchmark in **the distributed mode**:

Preparation: Provision 128 virtual machines, each with 2 vCPUs and 4GB RAM, connected over a LAN or WAN network. Additionally, set up a separate machine as a *jump server* that can access all 128 VMs via SSH. Ensure the network connectivity is functional.

Execution: Follow the instructions in the `benchmark` section of the `README.md` file. The script we prepared will automatically handle the setup and execution.

Note that to run this experiment, you only need the first three lines in the `run_all.sh` file.

Results: The results for each cases will be written to the `./output` directory. Use the provided Jupyter notebook (`./hack/read_data.ipynb`) to convert logs into CSV for performance comparisons. We expect to observe that the data reflect the Figure 3 and Table 2 in the paper. Note that in different runs, the computation time may vary slightly due to the non-deterministic nature of the network and system load. However, the overall linear trends should remain consistent.

If you are running the benchmark in **the local mode**:

Preparation: Prepare a single machine with at least 1TB of memory. This machine will run all the peers' workloads in a single thread, simulating the distributed network *locally*.

Execution: Run `bash ./hack/bench_hyperplonk.sh`. This script will automatically run the collaborative HyperPlonk and the monolithic prover in **local** mode, simulating the workload of $8l$ peers on a 2^n-gate circuit using a *single thread* locally.

Results: The output will be written to the `output` directory. The expected results are similar to the Figure 3 and Table 2 in the paper. Remember that in **local** mode, the actual computation time should be divided by $N = 8l$ where N is the number of peers, to obtain the average time per peer. After this division, the results should be similar to the distributed mode. For example, in the case of $l = 4$ and $n = 16$, the output should be similar to the following:

```
Start: Local HyperPlonk (thread
    ThreadId(1))
   ...
End: Local HyperPlonk (thread
    ThreadId(1)) 52.132s
Start: Local HyperPlonk++ (thread
    ThreadId(1))
   ...
End: Local HyperPlonk++ (thread
    ThreadId(1)) 65.256s
Start: Simulate Collaborative
    Hyperplonk++ (thread ThreadId(1))
   ...
Comm: (2298526, 3666742)
Comm: (2298526, 3666742)
End: Simulate Collaborative
    Hyperplonk++ (thread ThreadId(1))
     190.622s
```

Then each peer's average time is $\frac{190.6s}{4\times 8} = 5.9s$, while the monolithic prover's time is $65.3s$. Therefore, the simulated speedup is $\frac{65.3s}{5.9s} \approx 11$, which is consistent with the claims in the paper.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: DFS: Delegation-friendly zkSNARK and Private Delegation of Provers

Yuncong Hu [*] Pratyush Mishra [†] Xiao Wang [‡] Jie Xie [§] Kang Yang [¶] Yu Yu [‖]
Yuwen Zhang [**]

A Artifact Appendix

A.1 Abstract

This artifact appendix provides a roadmap for evaluators to test the functionality of our implementation. The artifact is based on the Arkworks framework and offers both single-prover and distributed setups.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We attest that we have thoroughly reviewed the ethics considerations as outlined in the conference call for papers, the detailed submission instructions, and the ethics guidelines document provided by the conference organizers. The research team has carefully evaluated the ethical implications of our work on DFS, ensuring that the research has been conducted in accordance with the highest ethical standards.

Our team has considered all potential ethical issues arising from this research, including the responsible disclosure of findings, the privacy implications of the technologies developed, and the potential for both positive and negative impacts on stakeholders. We have also proactively assessed the possible risks and mitigated them where necessary. We believe that our research was conducted ethically and in a manner that aligns with both the principles of beneficence and respect for persons as described in the Menlo Report.

Additionally, our next steps following publication have been carefully planned with ethical considerations in mind. We commit to following responsible procedures for the further dissemination and application of our findings, particularly in terms of sharing data and code in compliance with the conference's open science policy. We are prepared to engage with the broader community to address any ethical concerns that may arise as the research progresses.

Finally, we have also provided this additional Ethics Considerations and Compliance with the Open Science Policy section to ensure that all relevant ethical issues are transparent and addressed appropriately.

A.2.2 How to access

The artifact can be found in DOI 10.5281/zenodo.14677896.

A.2.3 Hardware dependencies

A local machine with the following specifications is sufficient for functionality testing:

- At least 10-core CPU
- 16 GB RAM

A.2.4 Software dependencies

- **Operating System:** Ubuntu 20.04+ (recommended) or MacOS
- **Rust Compiler:** nightly toolchain (v1.75+)
- **GNU Bash:** Required for running scripts

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Follow the instructions in the `README.md` file to set up the environment and dependencies. Note that we are using nightly toolchain for Rust.

[*] Shanghai Jiao Tong University, huyuncong@sjtu.edu.cn
[†] University of Pennsylvania, prat@seas.upenn.edu
[‡] Northwestern University, wangxiao@northwestern.edu
[§] Shanghai Jiao Tong University, xiejie1006@gmail.com
[¶] State Key Laboratory of Cryptology, yangk@sklc.org
[‖] Shanghai Jiao Tong University & Shanghai Qi Zhi Institute, yyuu@sjtu.edu.cn
[**] UC Berkeley, yuwen01@berkeley.edu

A.3.2 Basic Test

To verify the basic functionality of the artifact, follow these steps:
Setup and Compilation.

```
cargo build --release --examples
```

Running Functional Tests. Execute the following commands:

```
cargo test --release
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): For public delegations, DFS achieves logarithimic communication overheads.
(C2): For private delegations, DFS achieves logarithimic communication overheads for RSS-based implementations, and linear communication overheads for AddSS-based implementations.

A.4.2 Experiments

To run the experiments for example/rss_snark, example/ass_snark and example/snark, use the following commands:

```
./setup.sh
./work.sh
```

Before running the setup.sh, inst_folder folder must be created in the corresponding directory to store the generated files. The work.sh script will then run the tests.

mpirun $-$ n$*$ can be used to specify the number of cores to run. If it is changed, other parameters such as log-num-parties must also be changed accordingly.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: "SoK: Understanding zk-SNARKs: The Gap Between Research and Practice"

Junkai Liang[1,*], Daqi Hu[1,*], Pengfei Wu[2,*], Yunbo Yang[3], Qingni Shen[1,†], Zhonghai Wu[1,†]
[1]Peking University, [2]Singapore Management University, [3]East China Normal University
{ljknjupku, hudaqi0507}@gmail.com, pfwu@smu.edu.sg, yyb9882@gmail.com,
{qingnishen, wuzh}@pku.edu.cn

A Artifact Appendix

A.1 Abstract

As the inherent mathematical complexity has hindered the development of zk-SNARK fields, our artifact helps bridge the gaps between researchers, developers and users. The artifact consists of three main aspects: 1)sample code. We provide each sample code with a Docker image that allows users to conduct preliminary testing of these zkSNARK libraries. 2)tutorial. A detailed tutorial explaining the logical construction in the sample programs. 3)extended documentation (wiki book). This document outlines the circuit construction for each library's frontend, the zk-SNARK proving process in the backend, relevant APIs for library gadgets, and supported elliptic curves.

A.2 Description & Requirements

Our artifact contains several Docker virtual images that help practitioners test the efficiency of the three sample zk-SNARKs applications. The images can be run on any personal computer without installing the library.

A.2.1 Security, privacy, and ethical concerns

All evaluated zk-SNARK libraries are open-source and freely available on GitHub or their respective homepages. As such, this research does not involve any ethical concerns, as it does not include activities that could pose harm or risk to individuals or organizations.

A.2.2 How to access

Stable URL pointing to the latest version of our artifact: https://doi.org/10.5281/zenodo.14682405.

Stable URL for the feedback and dynamic changes: https://github.com/zk-lover/zk-sok.

A.2.3 Hardware dependencies

The codes in our artifact can be run on any general-purpose computer. The evaluation requires no special hardware features.

A.2.4 Software dependencies

We test our code on Ubuntu-20.04 with docker-26.1.4. The evaluation does not require specific OS or other software dependencies.

A.2.5 Benchmarks

None.

A.3 Set-up

We have packaged all the environments required for the experiments into Docker, so you don't need to prepare any environment except installing Docker.

A.3.1 Installation

Run the following instructions on any terminal in Linux operating system.

1. sudo apt update
2. sudo apt upgrade
3. apt-get install ca-certificates curl gnupg lsb-release
4. curl -fsSL http://mirrors.aliyun.com /docker-ce/linux/ubuntu/gpg
5. sudo apt-key add -
6. sudo add-apt-repository "deb [arch=amd64] http://mirrors.aliyun.com/docker-ce/linux /ubuntu $(lsb_release -cs) stable"
7. apt-get install docker-ce docker-ce-cli containerd.io

A.3.2 Basic Test

Taking Arkworks as an example, you can enter the /arkworks-lab directory and execute the following command to start Docker.

```
docker build -t arkworks .
docker run -it –rm arkworks
```

Then you can run our sample programs. For example:

```
cargo run –bin cubic_expression
```

A successful run will display output similar to the following:

```
Number of constraints: 3
Uncompressed pk size: 3984 bytes
Uncompressed vk size: 872 bytes
Total uncompressed size (pk + vk): 4856 bytes
Number of constraints: 3
Uncompressed proof size: 384 bytes
Proof is valid: true
Prove time: 74.518 milliseconds
Verify time: 51.528 milliseconds
```

You can use the following command to test the other two sample programs.

```
cargo run –bin rangeproof
cargo run –bin sha256
```

You can easily test the other eight libraries we evaluated using similar commands. We have packaged the code and environment into Docker, and you will see similar output.

A.4 Evaluation workflow

A.4.1 Major Claims

The main claim in our paper related to the artifact is in Section 5.4.3, and we explain the relation in detail as follows:

Our zk-sok open source repository evaluated a total of 9 libraries and implemented our three sample programs using the ZKP schemes they support, Cubic expression, Range proof and sha256. For different application scenarios, we recommend the best scheme along with its implementation. Groth16 is the best practice for applications that need a fast prover, a small proof size, and can tolerate a trust setup. gnark implements Groth16 more efficiently in Go, while snarkjs provides an implementation of Groth16 in Rust with more compatibility (using a DSL compiler). Plonk is the best practice for applications that need a transparent setup and are not sensitive to the slight increase in the proof size. For the widely used range proof, we recommend dalek, which is designed for range proof, specifically. This is proven by the experiment (E1-E9) described in [5.4 Experimental Evaluation] whose results are illustrated/reported in [Table 4].

A.4.2 Experiments

Our evaluation consists of 9 evaluations of the libraries. The instructions and descriptions are listed as follows:

(E1): *[gnark] [30 human-minutes + 5 compute-minutes +1GB disk]:*

Preparation: *No need to configure any environment except Docker*

Execution: Navigate to the /gnark directory:

```
cd /gnark
docker build -t gnark .
docker run -it –rm gnark
```

Navigate to the directory of the program you would like to run. Our examples are in the directories under /root. Run the following code to execute the examples.

```
cd testCubicequation_groth16
go build -o testCubicequation_groth16
./testCubicequation_groth16
```

(E2): *[libsnark] [30 human-minutes + 5 compute-minutes]:*

Preparation: Before building docker, execute:

```
git submodule update –init –recursive libsnark-lab/deps/libsnark
```

Execution: Navigate to the /libsnarklab directory:

```
cd /libsnarklab
Run docker:
docker build -t libsnark .
docker run -it –rm libsnark
```

Navigate to the directory of the program you would like to run. Our examples are at /app/src. Before you execute the sample program, you need run the following code to compile it.

```
mkdir build
cd build
cmake ..
make
```

Then you can run the following code to execute the sample program.

```
cd app/build/src
./cubic_expression
./range_proof
./sha256
```

(E3): *[libiop] [30 human-minutes + 5 compute-minutes]:*
Preparation: Before building docker, execute:

```
git submodule update –init –recursive libioplab/deps/libiop
```

Execution: Navigate to the /libioplab directory:

```
cd /libioplab
Run docker:
docker build -t libiop .
docker run -it –rm libiop
```

Navigate to the directory of the program you would like to run. Our examples are at /app/src. Before you execute the sample program, you need to run the following code to compile it.

```
mkdir build
cd build
cmake ..
make
```

Then you can run the following code to execute the sample program.

```
cd app/build/src
./cubic_aurora
./cubic_ligero
./cubic_fractal
./rangeproof_aurora
./rangeproof_ligero
./rangeproof_fractal
./sha256_aurora
./sha256_ligero
./sha256_fractal
```

(E4): *[snarkjs] [30 human-minutes + 5 compute-minutes + 10 compilation-hours]:*
Preparation: No need to configure any environment except Docker

Execution: Navigate to the /snarkjs directory:

```
cd /snarkjs
Run docker:
docker build -t snarkjs .
docker run -it –rm snarkjs
```

Navigate to the directory of the program you would like to run. Our examples are at /app. Enter a directory, you'll find 4 files: a .circom file, which is the circuit file and input.json file, which is the input file a start.sh file, which contains the full steps to run the program. a quick_start.sh file, which can be used when you have a final.ptau file. The final.ptau file contains the public parameters required by the circuit and is used to ensure the credibility of the circuit. It is a necessary file in the proof process. Snarkjs provides a command to generate this file. As the maximum constraints of the circuit increase, the file generation time is slower. Snarkjs officially provides final.ptau files ranging from 256 constraints to 256 mega constraints. You can choose to generate it yourself or download it. tips: To test sha256 requires a big ptau file, if you generate it yourself, it will take a long time. You can simply run the 'start.sh' to execute the examples.

```
./start.sh
```

Or when you have a final.ptau file, you can run the 'quick_start.sh' to execute the examples.

(E5): *[halo2] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker
Execution: Navigate to the /halo2 directory:

```
cd /halo2
Run docker:
docker build -t halo2 .
docker run -it –rm halo2
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin cubic_expression
cargo run –bin range_proof
cargo run –bin sha256
```

(E6): *[delak] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker.
Execution: Navigate to the /delak directory:

```
cd /delak
Run docker:
docker build -t delak .
docker run -it –rm delak
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin rangeproof
```

(E7): *[arkworks] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker

Execution: Navigate to the /arkworkslab directory:

```
cd /arkworkslab
Run docker:
docker build -t arkworks .
docker run -it –rm arkworks
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin cubic_expression
cargo run –bin rangeproof
cargo run –bin sha256
```

(E8): *[spartan] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker.
Execution: Navigate to the /spartan directory:

```
cd /spartan
Run docker:
docker build -t spartan .
docker run -it –rm spartans
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin cubic_expression
cargo run –bin rangeproof
cargo run –bin sha256
```

(E9): *[plonky2] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker.
Execution: If you want to run Cubic expression and Range proofs, navigate to the /plonky2 directory:

```
cd /plonky2
Run docker:
docker build -t plonky2 .
docker run -it –rm plonky2
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin cubic_expression
cargo run –bin range_proof
cargo run –bin sha256
```

Since plonky2 does not provide gadgets related to sha256 circuit construction, we seek an open source implementation of the forked plonky2 library and place it in our plonky2sha256 directory. If you want to run Sha256, navigate to the /plonky2sha256 directory:

```
cd /plonky2sha256
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: A limited technical background is sufficient for attack-defense tree acceptability

Nathan Daniel Schiele and Olga Gadyatskaya

A Artifact Appendix

A.1 Abstract

Study description

Attack-defense trees (ADTs) are a prominent graphical threat modeling method that is highly recommended for analyzing and communicating security-related information. Despite this, existing empirical studies of attack trees have established their acceptability only for users with highly technical (computer science) backgrounds while raising questions about their suitability for threat modeling stakeholders with a limited technical background. Our research addresses this gap by investigating the impact of the users' technical background on ADT acceptability in an empirical study.

Our Method Evaluation Model-based study consisted of $n = 102$ participants (53 with a strong computer science background and 49 with a limited computer science background) who were asked to complete a series of ADT-related tasks. By analyzing their responses and comparing the results, we reveal that a very limited technical background is sufficient for ADT acceptability. This finding underscores attack trees' viability as a threat modeling method.

Artifact description

This artifact includes the models (attack-defense trees) created by the participants in the study tasks, the study responses to perception questions (both Likert and short answer), the code used to statistically evaluate those survey responses, the study question text and images, the qualitative evaluation rubric for self-drawn ADTs, and the lecture plan and slides. With these artifacts, it should be possible to verify our results, organize training on attack-defense trees, and develop future studies of attack-defense tree acceptability.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our experiments are not destructive in any way. Our research involved human subjects, however all data in the artifact has been anonymized with no means to reconstruct original participants. Participants are identified with a unique code (`LT`/`HT`)`-##` in order to enable tracking of the same participant across different components (from image of ADT to survey responses). We described our handling of ethics concerns in Section 9 of our paper.

In order to perform the Holm-Bonferroni (HB) correction, all statistical tests need to be collected and corrected. For verification of the representativeness of our sample of the overall group of students, we statistically analyzed the overall course grade data of participants compared to non-participants. We ethically cannot release the underlying grade data for either participants and especially non-participants. We provided the original (non-corrected) results of performing these tests in the jupyter notebook `Statistics.ipynb` such that the results can be corrected (and subsequently affect the other corrections). As such, corrections can be independently performed but these tests cannot be independently performed due to ethical concerns.

A.2.2 How to access

The artifact is uploaded in its entirety to Zenodo and can be found using the following DOI: https://doi.org/10.5281/zenodo.14717342

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Our results were calculated with a Python virtual environment using Python version 3.12.1 with pip version 25.0.1. The following versions of jupyter and associated jupyter packages were used, however any version at least this version should sufficient:

- `IPython` – 8.31.0
- `ipykernel` – 6.29.5
- `jupyter_client` – 8.6.3
- `jupyter_core` – 5.7.2

A.2.5 Benchmarks

None.

A.3 Set-up

To run the statistics experiments, both the `Survey Data.csv` and `Statistics.ipynb` files should be downloaded and placed in the same directory. The Jupyter notebook should be run in a Python virtual environment with at least the versions of Python and the jupyter packages listed in Section A.2.4.

A.3.1 Installation

The first cell of the Jupyter notebook installs dependent packages. The cells are designed to be run in order.

A.3.2 Basic Test

A simple test would be to run all cells of `Statistics.ipynb`. If all cells run without error, the artifact is likely to be functioning correctly.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): There is no statistically significant difference in the actual effectiveness, perceived ease of use, perceived usefulness and intention to use of ADTs between users with a limited technical background and users with a highly technical background.

(C2:) Creative elements of the drawn of ADTs does not appear to be affected by the user's technical background.

A.4.2 Experiments

(E1): *[Statistical Testing Acceptability]: The statistical analysis of survey responses*

How to: For every hypothesis, we conducted the Brunner-Munzel test to assess if there is a statistically significant difference between the two groups. If no statistically significant difference is found, we performed the two one-sided test (TOST) to determine if there was a statistically significant equivalence between the two groups. All results are subsequently corrected using the Holm-Bonferroni correction. The measurement questions per hypothesis are as follows:

- $\mathbf{H}_1$
  - SS-Q3
  - leaf_node_avg
    - SS-Q2
    - SS-Q7
    - ADT1 # atk leaf nodes
  - def_node_avg
    - SS-Q8
    - SS-Q13
  - attack_vector_avg
    - SS-Q14
    - SS-Q18
  - LoA_avg
    - SS-Q9
    - SS-Q12
    - ADT1 # def leaf nodes
- $\mathbf{H}_{2-1}$
  - ADT2 # def leaf nodes
  - ADT2 # def nodes
  - ADT2 # atk leaf nodes
  - ADT2 # atk nodes
  - ADT2 # and (atk)
  - ADT2 #or (atk)
  - ADT2 LoA
- $\mathbf{H}_{2-2}$
  - ADT3 Cohesive
  - ADT3 Clear
  - ADT3 Concise
  - ADT3 Complete
- $\mathbf{H}_3$
  - ADT1 Multi parent nodes
  - ADT3 Multi parent nodes
  - ADT1 Multi refinement
  - ADT3 Multi refinement
  - ADT1 multi countermeasure
  - ADT2 multi countermeasure
  - ADT3 multi countermeasure
  - ADT1 single child (atk)
  - ADT2 single child (atk)
  - ADT3 single child (atk)
- $\mathbf{H}_4$
  - LS-ADT1-L1
  - SS-Q5
  - SS-Q10
  - SS-Q15
  - SS-Q19
- $\mathbf{H}_5$
  - LS-ADT3-L3
- $\mathbf{H}_6$
  - LS-ADT1-L5
  - LS-ADT2-L2
  - LS-ADT3-L1
- $\mathbf{H}_7$
  - LS-ADT2-L1
  - LS-ADT3-L2
  - SS-Q6
  - SS-Q11
  - SS-Q16
  - SS-Q20
- $\mathbf{H}_8$
  - LS-ADT3-W3 Yes
  - LS-ADT3-W3 Communication
  - LS-ADT3-W3 Analysis
  - LS-ADT3-W5 Yes
- $\mathbf{H}_9$
  - ADT3 # def leaf nodes
  - ADT3 # def nodes

- ADT3 # atk leaf nodes
- ADT3 # atk nodes
- ADT3 # and (atk)
- ADT3 #or (atk)
- ADT3 LoA
- ADT3 and:or ratio

Preparation: Download the `Survey Data.csv` file. Additionally, download the `Statistics.ipynb` file and place it in the same folder. The columns of the survey data are labeled with the measurement questions. The data should be collected into the comparison groups (HT vs. LT). For hypothesis H_1, preprocessing the data to create average correctness values is required.

Execution: For each of the hypotheses listed above, perform the Brunner-Munzel test. If the test is not statistically significant (p-value > 0.05), perform the two one-sided test (TOST). Correct all tests using the Holm-Bonferroni correction, and perform new TOST tests if previous results are determined to be no longer statistically significant.

Note: This is what the code in the jupyter notebook `Statistics.ipynb` does. These tests will be run in the manner described here by running all cells o the notebook.

Results: The corrected p-values should align with the results in the original paper presented in Tables 2–7. These results are used to justify claims C1 and C2.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Prompt Obfuscation for Large Language Models

David Pape[1], Sina Mavali[1], Thorsten Eisenhofer[2,3], Lea Schönherr[1]

[1]CISPA Helmholtz Center for Information Security
[2]Berlin Institute for the Foundations of Learning and Data (BIFOLD)
[3]Technische Universität Berlin

A Artifact Appendix

A.1 Abstract

This artifact provides the source code and framework for a prompt obfuscation method that safeguards the intellectual property of system prompts by creating functionally equivalent, unintelligible representations. It implements two techniques: a 'hard' method in the discrete token space and a 'soft' method in the continuous embedding space. The artifact includes code to evaluate obfuscated prompts, run deobfuscation attacks (e.g., prompt extraction, projection, and fluency optimization), compare against a LoRA finetuning baseline, and automatically download all required public models and datasets.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The provided artifact poses no security or privacy risks to the evaluator. The experiments exclusively use publicly available datasets and models downloaded from HuggingFace. Furthermore, the artifact does not perform any destructive file operations outside of its own designated output directories.

A.2.2 How to access

The artifact is available on Zenodo: `https://doi.org/10.5281/zenodo.15601914`.

A.2.3 Hardware dependencies

A GPU is required for running this artifact. We recommend using an environment with an NVIDIA GeForce RTX 3090 or more powerful GPU. Around 35GB of disk space is needed to store the main language model, datasets, and other supporting models downloaded from Hugging Face. All experiments reported in the paper were conducted on a single NVIDIA A100 GPU with 40 GB of VRAM.

A.2.4 Software dependencies

All experiments were conducted in a conda environment with Python 3.12.7 on Ubuntu 22.04.4, though any operating system supporting Python works. All necessary dependencies are listed in `requirements.txt` in the artifact, with detailed installation instructions outlined in Section A.3.1.

A.2.5 Benchmarks

The artifact relies on several publicly available models, datasets, and a set of curated prompts, all of which are essential for reproducing the experiments reported in the paper.

- **Models**: The primary model used is Llama 3.1-8B, which is automatically downloaded from Huggingface. Please note that this is a gated model requiring explicit agreement to Meta's Community License. Supporting models for evaluation are also downloaded automatically from the Hugging Face Hub.
- **Datasets**: Our experiments utilize four public datasets: TruthfulQA, TriviaQA, CNN/DailyMail, and Samsum. The provided data loaders automatically download and process the correct versions of these datasets from the Hugging Face Hub.
- **Prompt Extraction Queries**: The set of 105 prompt extraction queries from Zhang et al. used for the attack evaluation in Section 6.1 is included directly within the artifact at `extraction_prompts/gpt4_generated.json`.

A.3 Set-up

A.3.1 Installation

First, download the artifact from Zenodo and extract it. We recommend creating a new virtual environment with Python 3.12.7 (e.g., using conda):

```
conda create -n prompt_obfuscation python=3.12.7
conda activate prompt_obfuscation
```

Next, install all necessary dependencies:

```
pip install -r requirements.txt
```

After account creation and access is granted to Llama 3.1, log in using the following command:

```
huggingface-cli login
```

Note: All models and datasets are downloaded to `~/.cache/`. To change the default path, you can run:

```
export HF_HOME="/new/path"
export SENTENCE_TRANSFORMERS_HOME="/new/path"
export NLTK_DATA="/new/path"
```

A.3.2 Basic Test

We provide the script `basic_test.py` to perform a lightweight functionality test that checks the correct setup of all core software components and hardware dependencies. To run the test, execute the following command from the project's root directory.

```
python3 basic_test.py
```

The script will print detailed progress for each step. A successful execution concludes with the final confirmation message: "All Basic Tests Passed Successfully! The environment is set up correctly and all core components are functional."

A.4 Evaluation workflow

A.4.1 Major Claims

Our paper makes the following major claims:

(C1): Hard prompt obfuscation preserves the utility of the original prompt but is susceptible to information leakage. This is proven by experiments (E1) and (E2), described in Section 5.1, with results in Tables 1 and 2.

(C2): Soft prompt obfuscation effectively preserves the functionality of conventional system prompts across diverse tasks and in a real-world scenario. This is proven by experiments (E3) and (E4), described in Sections 5.2 and 5.3, with results in Tables 4, 5, 11, and 12.

(C3): Prompt obfuscation is a competitive alternative to fine-tuning, achieving similar utility with significantly lower storage overhead. This is proven by experiment (E5), described in Section 5.4, with results in Table 6.

(C4): Obfuscated soft prompts are robust against prompt extraction and direct projection attacks. This is proven by experiments (E6) and (E7), described in Sections 6.1 and 6.2, with results in Tables 7 and 8.

(C5): Under a white-box threat model, a fluency optimization attack can partially recover human-readable fragments. This is proven by experiments (E8) and (E9), described in Section 6.3, with results in Tables 9 and 10.

A.4.2 Experiments

The following experiments provide the steps to reproduce the evidence for the major claims. Each is demonstrated with a representative example. To facilitate quick verification of computationally intensive experiments, precomputed results for specific styles are provided in the artifact's `precomputed_results/` directory. All other results reported in the paper can be reproduced by running the provided scripts with different parameters (e.g., `--style`, `--dataset_name`). For a comprehensive list of arguments, please refer to the main `README.md`. A complete list of available style identifiers is defined in `src/style_prompts.py` and listed in Appendix A of the paper. Note: The estimated compute times and VRAM requirements may vary slightly depending on the specific style and dataset chosen.

(E1): [Hard Prompt Obfuscation Utility] [12 compute-hours, 23GB VRAM]: This experiment evaluates obfuscated hard prompt utility against baselines.
How-to: We provide scripts for "Full", "Style", and "Task" scenarios. The "Task" scenario script is run once, while others are run for each style.
Execution: To reproduce the "Full" scenario for the 'pirate' style:

```
./bash_scripts/hard_prompt_obfuscation_full.sh
    --style pirate
```

For the "Style" and "Task" scenarios, run `hard_prompt_obfuscation_style.sh --style <style>` and `hard_prompt_obfuscation_task.sh`, respectively.
Results: Outputs are in their respective subdirectories (e.g., `results/hard_pirate_full/`). The key files to verify against Table 1 are:

- `best_candidate_scores.json` ('obf' column)
- `blank_output_scores.json` ('blank' column)
- `original_output_scores.json` ('original' column)

(E2): [Hard Prompt Information Leakage] [30 compute-seconds, 1.5GB VRAM]: This experiment quantifies information leakage from the best hard prompt.
How-to: This script requires the results directory from a corresponding (E1) experiment.
Execution: To evaluate leakage for the 'Full' 'pirate' scenario from (E1):

```
./bash_scripts/compare_hard_prompt_leakage.sh
    --results_dir results/hard_pirate_full
```

Results: The key files to verify against Table 2 are `obf_sys_prompt_scores.json` ('obf' column) and `rand_sys_prompt_scores.json` ('rand' column).

(E3): [Soft Prompt Obfuscation Utility] [4 compute-hours, 25GB VRAM]: This experiment evaluates obfuscated soft prompt utility against baselines.
How-to: Similar to (E1), we provide scripts for "Full",

"Style", and "Task" scenarios, which take `--style` and `--dataset_name` arguments (except for the task script, which only needs `--dataset_name`).
Execution: To reproduce the "Full" scenario for the ‘pirate‘ style on ‘truthfulqa‘:

```
./bash_scripts/soft_prompt_obfuscation_full.sh
    --style pirate --dataset_name truthfulqa
```

Results: The output directory (e.g., `results/soft_pirate_truthfulqa_full/`) will contain the same set of score files as in (E1), used to verify results in Tables 4, 11, and 12.

(E4): [Case Study] [4 compute-hours, 25GB VRAM]: This experiment demonstrates soft prompt obfuscation on a real-world prompt.
How-to: The experiment is encapsulated in a single script that requires no command-line arguments.
Execution: Execute the script:

```
./bash_scripts/soft_prompt_obfuscation_case_study.sh
```

Results: The output directory `results/soft_manga_miko_case_study/` contains score files (same as (E1)) to verify the results in Table 5.

(E5): [Comparison to LoRA Finetuning] [1 compute-hour, 13GB VRAM]: This experiment fine-tunes a LoRA adapter as a baseline.
How-to: We provide scripts for "Full", "Style", and "Task" scenarios, mirroring the structure of (E3).
Execution: To reproduce the "Full" scenario for ‘pirate‘ on ‘truthfulqa‘:

```
./bash_scripts/run_finetuning_full.sh --style
    pirate --dataset_name truthfulqa
```

Results: The output directory (e.g., `results/finetuning_pirate_truthfulqa_full/`) contains `best_adapter_scores.json`. These scores correspond to the ’finetune’ column in Table 6 and can be compared with the ’obf’ scores from (E3).

(E6): [Robustness Against Prompt Extraction] [15 compute-minutes, 14GB VRAM]: This experiment evaluates robustness against prompt extraction attacks.
How-to: This script requires the results directory from a soft prompt obfuscation run (E3).
Execution: To test the ‘pirate‘ style on ‘truthfulqa‘ from (E3):

```
./bash_scripts/run_prompt_extraction_attack.sh
    --results_dir
    results/soft_pirate_truthfulqa_full
```

Results: The console output reports the number of successful extractions for both obfuscated and conventional prompts, corresponding to the data in Table 7.

(E7): [Robustness Against Projection Attack] [2 compute-minutes, 10GB VRAM]: This experiment evaluates deobfuscation via direct projection to the token space.
How-to: This script requires the results directory from a soft prompt obfuscation run (E3).
Execution: To test the ‘pirate‘ style on ‘truthfulqa‘ from (E3):

```
./bash_scripts/run_projection_attack.sh
    --results_dir
    results/soft_pirate_truthfulqa_full
```

Results: The key files to verify against Table 8 are `euclidean_sys_prompt_scores.json`, `cosine_sys_prompt_scores.json`, and `random_sys_prompt_scores.json`.

(E8): [Soft Fluency Deobfuscation Attack] [3.5 compute-hours, 30GB VRAM]: This experiment runs the "soft" fluency optimization attack.
How-to: The script requires the results directory from a soft prompt obfuscation run (E3).
Execution: Using the ‘pirate‘ style on ‘truthfulqa‘ from (E3):

```
./bash_scripts/run_fluency_deobfuscation_soft.sh
    --results_dir
    results/soft_pirate_truthfulqa_full
```

Results: The output directory will contain the similarity scores for the deobfuscated prompt and the random baseline. The key files to verify against Table 9 are `best_deobf_sys_prompt_soft_scores.json` (’deobf’ column) and `random_sys_prompt_scores.json` (’rand’ column).

(E9): [Hard Fluency Deobfuscation Attack] [7 compute-hours, 31GB VRAM]: This experiment runs the "hard" fluency optimization attack.
How-to: The script requires the results directory from a soft prompt obfuscation run (E3).
Execution: Using the ‘pirate‘ style on ‘truthfulqa‘ from (E3):

```
./bash_scripts/run_fluency_deobfuscation_hard.sh
    --results_dir
    results/soft_pirate_truthfulqa_full
```

Results: The output directory will contain the similarity scores for the deobfuscated prompt and the random baseline. The key files to verify against Table 10 are `best_deobf_sys_prompt_hard_scores.json` (’deobf’ column) and `random_sys_prompt_scores.json` (’rand’ column).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: TwinBreak: Jailbreaking LLM Security Alignments based on Twin Prompts

Torsten Krauß
University of Würzburg

Hamid Dashtbani
University of Würzburg

Alexandra Dmitrienko
University of Würzburg

A Artifact Appendix

A.1 Abstract

This artifact accompanies the USENIX paper "TwinBreak: Jailbreaking LLM Security Alignments with Twin Prompts". It delivers a complete, reproducible implementation for removing safety alignments from downloaded large language models (LLMs), such as open-source models on Hugging Face, while safeguarding their utility. As the basis for TwinBreak, this artifact also contains the TwinPrompt dataset, a collection of one hundred prompt pairs with one harmful and one harmless prompt that yield high structural and content similarity. Feeding these prompts through a target model, TwinBreak traces intermediate activations to rank each parameter by its contribution to safety enforcement. Feeding two harmless prompts allows one to identify parameters mandatory for utility. Parameters deemed critical to safety are pruned at inference time unless they also prove indispensable for utility, thereby disabling safety alignment without degrading utility. The artifact bundles the TwinBreak source code, the TwinPrompt dataset, and scripts needed to reproduce experiments that demonstrate successful alignment removal and consequent jailbreaks on recent LLMs.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Executing this artifact poses no risk.

A.2.2 How to access

The artifact can be accessed via GitHub and Zenodo.

- GitHub: https://github.com/tkr-research/twinbreak
- Zenodo: https://doi.org/10.5281/zenodo.15591819

A.2.3 Hardware dependencies

The experiments in this artifact run reliably on Unix-based servers equipped with a sufficiently powerful NVIDIA GPU with at least 48 GB of memory. More powerful GPUs and greater memory capacities will lead to faster runtimes. In addition, approximately 120 GB of disk space is required for downloading large language models (LLMs) from Hugging Face. The codebase and experimental outputs occupy roughly 100 MB. We also recommend using a Conda environment for managing dependencies, which consumes around 7 GB of disk space.

Minimum System Requirements:

- NVIDIA GPU with at least 48 GB of memory
- 130 GB of available disk space

Hint 1. Reproducing the full experimental suite from the paper, including models with up to 70 billion parameters, demands significantly more resources, both GPU memory and disk space. For reference, our experimental setup used an Intel Xeon Gold 6526Y CPU (16 cores, 64 threads), 256 GB of RAM, four NVIDIA L40S GPUs (each with 48 GB of GDDR6 memory), and a 7 TB HDD. Configurations for these large-scale experiments are included for completeness but are not part of the prepared demonstration for this artifact, focusing on TwinBreak's core functionality. Users may select the respective model architectures in the configuration file to execute them, but can safely skip them for this artifact.

Hint 2. The most memory-intensive aspect of the artifact is the generation of LLM responses during utility and safety evaluations, not the TwinBreak attack itself. By default, a batch size of 20 is used for inference during benchmark executions. To reduce runtime, this can be increased up to 100, provided sufficient GPU memory is available. If the available GPU memory is less than 48 GB, the batch size may need to be reduced to fit the model, though this will come at the cost of longer runtimes.

A.2.4 Software dependencies

The experiments are designed to run on a Unix-based operating system, such as Debian, which serves as our reference platform. The environment requires Python 3.10 and PyTorch 2.7.0, with CUDA-enabled access to NVIDIA GPUs. While we recommend using CUDA 12.6, we also provide instructions for alternative CUDA versions.

We suggest using `Miniconda` to create an isolated Conda environment, which simplifies setup and ensures proper de-

pendency management. All required packages can be installed via the pip package manager.

Required Python Packages:

- torch 2.7.0
- transformers 4.44.2
- lm_eval 0.4.7
- StrongREJECT
- python-dotenv 0.9.9
- pyyaml 6.0.2

Hint. We provide specific package versions to ensure a smooth and consistent reproduction process. In particular, newer versions of transformers may lead to issues related to torch._dynamo, which attempts to compile Python model code into a single optimized computation graph. If the model's code involves frequent changes in shape, type, or control flow, such as from dynamic forward() logic or the use of hooks, Dynamo may recompile the graph repeatedly, eventually reaching the default limit of 8 recompilations. This issue arises due to the pruning implementation, which relies on forward hooks. While it is possible to circumvent the problem by disabling TorchDynamo Just-In-Time (JIT) Compilation using the TORCHDYNAMO_DISABLE=1 environment flag, doing so can significantly increase execution time. Therefore, we recommend using the package versions and settings specified in the paper.

Hugging Face Access. To download the required LLMs from Hugging Face, users must have a Hugging Face account that has accepted the respective model license agreements. An access token associated with this account is necessary and must be used within the project to authenticate and enable model downloads.

The following models, used in the core experiments of this artifact, require license agreement on the respective Hugging Face website and subsequent access token authorization:

- https://huggingface.co/meta-llama/Llama-2-7b-chat-hf
- https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
- https://huggingface.co/google/gemma-2-9b-it
- https://huggingface.co/Qwen/Qwen2.5-7B-Instruct
- https://huggingface.co/meta-llama/Llama-2-13b-chat-hf
- https://huggingface.co/Qwen/Qwen2.5-3B-Instruct
- https://huggingface.co/meta-llama/Llama-Guard-3-8B
- https://huggingface.co/google/gemma-2b

Executing experiments with additional models may require accepting further license agreements.

A.2.5 Benchmarks

The artifact contains three dataset categories: one for measuring utility, another for assessing safety alignment, and the newly introduced TwinPrompt dataset used during the TwinBreak attack.

Utility Datasets. The artifact uses five dataset to utility analysis. All datasets can be easily downloaded and are open-source. The five datasets are: 1) OpenBookQA, which tests an LLM's reasoning and knowledge absorption capability with a focus on preliminary scientific topics. 2) ARC-Challenge that targets more complex science questions. 3) HellaSwag which asks the LLM to choose the most plausible continuation scenario given a partial sentence or scenario. 4) RTE, which evaluates with whether a hypothesis can be inferred from a premise. 5) WinoGrande evaluates an LLM's common sense and contextual understanding. The datasets are downloaded and used on the fly via the lm_eval Python package.

Safety Alignment Datasets. The artifact uses four datasets comprising harmful prompts to assess LLMs' security and resilience against potential misuse. These prompts are designed to mimic malicious interactions. All datasets are open-source and are already part of the repository formatted as JSON files. The four datasets and respective links to the original dataset files are: 1) AdvBench, which contains 520 harmful prompts. 2) HarmBench, an improved version of AdvBench with 400 harmful prompts. Following related work (as discussed in the paper), we only use 200 prompts from HarmBench. 3) Jailbreakbench, which contains 100 pairs of harmful and harmless prompts. However, the pairs are not twins as in our dataset. 4) StrongREJECT, which contains 313 harmful prompts trying to address shortcomings of the previous datasets.

New TwinPrompt Dataset. Finally, this artifact publishes the new dataset, TwinPrompt, which consists of one hundred prompt pairs with one harmful and one harmless prompt that yield high structural and content similarity. The dataset is also part of this repository and in JSON format and can be found under twinbreak/dataset/json/twinprompt.json.

A.3 Set-up

A.3.1 Installation

It is assumed that the setup begins with a newly created user account on a Unix-based server, such as a Debian system.

To ensure reproducibility and consistent package management, we use Miniconda to manage the Python environment.

While it is possible to install the dependencies globally without Miniconda, we recommend using Miniconda as described below. If Miniconda is skipped, users must manually install Python 3.10 and ensure all dependencies are correctly resolved.

Hint: When copying multi-line commands from the PDF, we recommend first pasting them into a plain text editor (e.g., Notepad) to remove any unintended line breaks before executing them in the terminal. The symbol ↪ indicates that the command continues on the next line without any added space; it is used solely for formatting purposes due to line length constraints in the PDF layout. Alternatively, the commands can be copied from the README file in the GitHub repository.

Hugging Face Token: If you already have access to a Hugging Face access token with read permissions, you can use it directly. Otherwise, follow the steps below to create your own token:

1. Create an Hugging Face account. Such an account can be created for free.
2. To generate an access token, visit https://huggingface.co/settings/tokens, click "Create new token", select the "Read" token type, and assign a name to the token. Once created, copy the token for later use.
3. To get access to the models, please follow the license agreement instructions provided on the Hugging Face links provided under Section A.2.4. After confirming the agreements, it may take up to an hour for your access status to change from "pending" to "accepted."

Miniconda Installation: If `Miniconda` is not already installed on the server, follow the steps below to install Miniconda.

1. Download the Miniconda installer:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-
    ↪ latest-Linux-x86_64.sh
```

2. Run the installer:

```
bash Miniconda3-latest-Linux-x86_64.sh
```

During installation, follow these steps:

- Press `ENTER` to view the license terms.
- Press `SPACE` to scroll to the end of the license terms.
- Type `yes` to accept the license agreement.
- Press `ENTER` to confirm the default location for conda environments.
- Type `yes` to enable automatic activation of conda on shell startup.

Afterward, restart your shell, such that conda is activated. The shell should look like this showing that the base conda environment is activated:

```
(base) user@server:/home/user$
```

Environment Setup

1. Create and activate the Conda environment:

```
conda create --name twinbreak python=3.10
```

During the installation process enter y to install the new packages.

```
conda activate twinbreak
```

Afterward, your shell should look like this, showing that the new twinbreak environment is activated:

```
(twinbreak) user@server:/home/user$
```

2. Identify your CUDA version (if applicable):

```
nvidia-smi
```

The CUDA version appears in the top-right corner of the table output.

3. Install PyTorch corresponding to your CUDA version:
   - For CUDA 11.8:

```
pip install torch --index-url https://download.
    ↪ pytorch.org/whl/cu118
```

   - For CUDA 12.6 (default if compatible):

```
pip install torch
```

   - For CUDA 12.8 or later:

```
pip install torch --index-url https://download.
    ↪ pytorch.org/whl/cu128
```

   If you have a different CUDA version, you should try out the next smaller PyTorch version.

4. Install the remaining dependencies:

```
pip install transformers==4.44.2
pip install lm_eval==0.4.7
pip install git+https://github.com/dsbowen/
    ↪ strong_reject.
    ↪ git@e286f0da86d92c929a6fda20a9992f28c5969044
pip install dotenv==0.9.9
pip install pyyaml==6.0.2
```

Artifact Retrieval and Initialization

1. Navigate to the desired directory where the project should be stored and clone the repository:

```
git clone https://github.com/tkr-research/twinbreak.
    ↪ git
```

2. Enter the project root:

```
cd twinbreak
```

3. Run the setup script, providing your Hugging Face access token and the model storage path. Therefore, replace `<HUGGING_FACE_TOKEN>` and `<STORE_MODEL_DISK_PATH>` with the real values, e.g., `hf_xxxYOURTOKENxxx` and `\home\user\.cache\huggingface`. Note, that the terminal user needs read and write permissions to the selected directory. The two values will be saved in a `.env` file and used to set necessary environment variables. Additionally, the script will configure the project root as part of the Python path.

```
source setup.sh --hf-token <HUGGING_FACE_TOKEN> --
    ↪ store-model-disk-path <STORE_MODEL_DISK_PATH>
```

The terminal should output the following:

```
.env file created and values injected:
  - HF_TOKEN
  - STORE_MODEL_DISK_PATH
PYTHONPATH set to /home/user/twinbreak
```

A.3.2 Basic Test

To run a simple functionality test, execute the following commands.

1. Navigate to the experiments directory:

```
cd experiments
```

2. Execute the test script to verify the setup:

```
python experiment_test.py
```

3. If successful, the following message will be displayed:

```
The system is set up to run experiments!
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): TwinBreak effectively removes safety alignment from open-source large language models (LLMs) with minimal impact on their utility, as demonstrated on the LLaMA 2 (7B) model. This is proven by experiment (E1) in this artifact, which is also reported in Section 4.2 of the paper, with results reported in the first rows of Tables 2–5 for safety alignment removal, and Tables 15–19 for utility preservation.

(C2): TwinBreak demonstrates effectiveness across diverse model architectures and vendors, as shown on Llama 3.1 (8B), Gemma 2 (9B), and Qwen 2.5 (7B). This is proven by experiment (E2) in this artifact, which is also reported in Section 4.2 of the paper, with results reported in the remaining rows of Tables 2–5 for safety alignment removal, and Tables 15–19 for utility preservation.

(C3): TwinBreak also proves effective across varying model sizes, as demonstrated on the larger LLaMA 2 (13B) and the smaller Qwen 2.5 (3B). This is proven by experiment (E3) in this artifact, which is also reported in Section 4.6 of the paper, with results reported in rows two and six of Table 8.

A.4.2 Experiments

Hint 1. The reported experiment runtimes were obtained using a single NVIDIA L40s GPU with 48 GB of memory. Actual runtimes may vary depending on the number of GPUs, GPU model, and available memory. By default, the provided code will automatically utilize all available GPUs on the server.

Hint 2. If you wish to restrict execution to a specific subset of GPUs, you will need to modify the command used to run the Python scripts. For example, to use only the GPUs with indices 1 and 2, the command would be:

```
CUDA_VISIBLE_DEVICES=1,2 python experiment_test.py
```

*Hint 3.*Given the potentially long runtime of the experiments, we recommend using the `screen` tool to prevent interruptions in case the connection to the server is lost. This helps ensure that experiments continue running even if the terminal session is disconnected. For usage instructions, please refer to the `screen` manual page. However, this step is optional for executing the artifact.

Below, three experiments are described, each proving one major claim of this project.

(E1): [General Functionality] [1 human-minute + 2 compute-hour + 35GB disk]: In this experiment, we attack the LLaMA 2 (7B) model with TwinBreak and evaluate the jailbreak success as well as the utility with all benchmarks. The results prove the major claim (C1).

Preparation: Follow all instructions under Section A.3. The terminal should reside in the `twinbreak\experiments` folder.
Execution: Execute the script for the first experiment.

```
python experiment_1.py
```

Results: The experiment logs its results both to the terminal and to a log file located at `twinbreak\results\experiment_1\log\log0.txt`. At the end of the output, a summary of the results is presented in two tables. The first table reports the utility benchmark results and corresponds to the first rows of Tables 15–19 in the paper. The second table presents the safety benchmark results and can be compared to the first rows of Tables 2–5. For clarity, the relevant reference table from the paper is also indicated alongside each output table in the terminal.
We expect the reported values to fall within a 1–5% range of those published in the paper. Additionally, the runtime of the attack is displayed beneath the tables and can be compared to the values in Table 7. Note that runtime is highly dependent on hardware. For example, while the paper reports a runtime of 162 seconds, our server achieved significantly faster results (16 seconds), as the experiment was run on Kaggle.

(E2): [Model Architecture Independence] [3 human-minute + 7 compute-hour + 114GB disk]: In this experiment, we repeat experiment (E1) with different model architectures from different vendors, namely Llama 3.1 (8B), Gemma 2 (9B), and Qwen 2.5 (7B). The results prove the major claim (C2).
Preparation: Follow all instructions under Section A.3. The terminal should reside in the `twinbreak\experiments` folder.
Execution: We provide one script for each of the models. Execute the scripts individually. To execute TwinBreak for Qwen 2.5 (7b), execute the following command. [1 human-minute + 2 compute-hour + 37GB disk]

```
python experiment_2_1.py
```

To execute TwinBreak for Gemma 2 (9b), execute the following command. [1 human-minute + 3 compute-hour + 40GB disk]

```
python experiment_2_2.py
```

To execute TwinBreak for LLaMA 3.1 (8b), execute the following command. [1 human-minute + 2 compute-hour + 37GB disk]

```
python experiment_2_3.py
```

Results: As with experiment (E1), each script generates terminal output and corresponding log files stored in the appropriate results folder, for example, `twinbreak\results\experiment_2_1\log\log0.txt` for `experiment_2_1.py`. The output can be compared to the utility and safety benchmarks in Tables 15–19 and Tables 2–5, respectively. As before, we expect the reported values to be within a 1–5% range of those presented in the paper.

(E3): [Model Size Independence] [2 human-minute + 4 compute-hour + 75GB disk]: In this experiment we attack models with different model sizes, namely LLaMA 2 (13b) and Qwen 2.5 (3b), using TwinBreak. We evaluate the jailbreak success with the newest benchmark (StrongREJECT) and the utility with HellaSwag. The results prove the major claim (C3).
Preparation: Follow all instructions under Section A.3. The terminal should reside in the `twinbreak\experiments` folder.
Execution: We provide one script for each of the models. Execute the scripts individually. To execute TwinBreak for LLaMA 2 (13b), execute the following command. [1 human-minute + 3 compute-hour + 47GB disk]

```
python experiment_3_1.py
```

To execute TwinBreak for Qwen 2.5 (3b), execute the following command. [1 human-minute + 1 compute-hour + 28GB disk]

```
python experiment_3_2.py
```

Results: As with experiment (E1), each script generates terminal output and corresponding log files stored in the appropriate results folder, for example, `twinbreak\results\experiment_3_1\log\log0.txt` for `experiment_3_1.py`. The output can be compared to the utility and safety benchmark results in Table 8. Specifically, the `AVG Degradation` value in the utility benchmarks corresponds to the `TwinBreak Utility` column for the respective model in Table 8, while the StrongREJECT score in the `Iteration 5` column reflects the `TwinBreak ASR` column in Table 8. As with previous experiments, we expect the reported values to fall within a 1–5% range of those presented in the paper.

Hint. In order to clean up after executing the artifact, follow the cleanup instructions in the README file of the GitHub repository.

A.5 Notes on Reusability

The core functionality of *TwinBreak* is implemented in `TwinBreak.py`. Configuration options and hyperparameters are managed via YAML files, with `twinbreak_default_settings.yaml` serving as the default configuration. Each parameter is documented in the paper, the YAML file itself, and the corresponding Python class `TwinBreakConfig.py`.

To evaluate TwinBreak on benchmark tasks, use the

`TwinBreakAndEval.py` wrapper, which extends the core functionality of `TwinBreak.py` with evaluation capabilities.

Customizing Experiments: There are two main types of configuration files:

- **TwinBreak configuration** (e.g., utility weight parameters): Default: `twinbreak_default_settings.yaml`
- **Experiment configuration** (e.g., selected model, benchmarks, batch size): Default: `experiment_default_settings.yaml`

The entry point for running a complete experiment is `ExperimentExecutor.py`, which provides a `run()` function to execute the pipeline.

Hyperparameter Study: Configuration files used for the hyperparameter analysis (as reported in Table 20 of the paper) are available in the `hyperparameter_study` folder, with `H1.yaml` being the first file.

Extending to New Models: To apply *TwinBreak* to models not currently supported, implement a new subclass of the `AbstractModel.py` class. Then, reference your custom model identifier in the experiment configuration file.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: SoK: So, You Think You Know All About Secure Randomized Caches?

Anubhav Bhatla*, Hari Rohit Bhavsar*, Sayandeep Saha, Biswabandan Panda
Indian Institute of Technology Bombay

A Artifact Appendix

A.1 Abstract

The artifact is used to evaluate the security of various cache designs discussed in the paper. We provide the source files and scripts required to reproduce all security results (Figures 3–19 and Table 2). A comprehensive README file is also included to facilitate easy reproduction of the results.

A.2 Description & Requirements

This section lists all the hardware and software dependencies required for the experimental setup to run all security simulations. Given below is a description of the important directories and scripts provided in the artifact:

- **cache-model:** An extended version of the behavioral cache simulation model. This model is used to generate results for Figures 3–14, 18–19, and Table 2.
- **cachefx:** An extended version of the CacheFX simulator used to generate results for Figures 15–16.
- **low-occupancy:** An extended simulation model for our low-occupancy-based attack simulations. This model is used to generate results for Figure 17.
- **`buildAll.sh`:** Script used to build all source files for the cache model, CacheFX, and low-occupancy simulators.
- **`genAllFigs.sh`:** Script used to run all experiments and generate PDF files for Figures 3–19.
- **`genTable.sh`:** Script used to run experiments and generate Table 2.
- **`requirements.txt`:** A list of the Python libraries needed for the smooth functioning of all scripts.
- **`README.md`:** Provides a description of the directories and detailed steps to build projects and run experiments to reproduce results from this work.

*These authors contributed equally to this work

A.2.1 Security, privacy, and ethical concerns

Our study does not introduce any novel or previously unknown attacks. Instead, it focuses exclusively on attacks that have already been disclosed and documented in prior work. All the attacks discussed in our evaluation are well-established and widely recognized within the community. The experiments were carried out using software-based, open-source simulators to replicate the required conditions and analyze system behaviors. No real-world systems were accessed, and no vulnerabilities were discovered, exploited, or tested on live or operational systems. Therefore, there are no ethical concerns in this work.

A.2.2 How to access

Our artifact is available through Zenodo. The artifact can be accessed at https://doi.org/10.5281/zenodo.15529618.

A.2.3 Hardware dependencies

We implemented and evaluated our artifact on a server with the Intel Xeon Gold 6342 CPU, 128GB of RAM, and 4TB of disk storage. To effectively parallelize simulations and ensure timely completion, we recommend utilizing a large number of CPU cores, ideally distributed across multiple servers.

A.2.4 Software dependencies

We test our artifact on a system running Ubuntu 20.04.6. However, any operating system which supports C++ and Python is sufficient to run our experiments. The Python packages required have been listed in the `requirements.txt` file. We also require docker and the C++ boost library for our simulations. The instructions to install this library and the Python libraries can be found in the `README.md` file.

A.2.5 Benchmarks

Our experiments evaluate the eviction rate of cache designs, number of LLC evictions required for eviction-set-generation, and the guessing entropy for key recovery. These experiments are self-sufficient and do not require any benchmarks for evaluation.

A.3 Set-up

A.3.1 Installation

We require that python, pip3, and docker are already installed on the system. For installing required Python libraries, run:

```
$ pip3 install -r requirements.txt
```

We also require the C++ boost library, installed using:

```
$ sudo apt install libboost-all-dev
```

The source files can be compiled using the following:

```
$ bash buildAll.sh build
```

A.3.2 Basic Test

An easy way to check if the simulation set-up is complete is to generate all figures and the table using our existing results. These can be generated using:

```
$ bash genAllFigs.sh 1
$ bash genTable.sh 1
```

If the set-up is complete, 17 PDF files will appear, named `figurex.pdf`, where x ranges from 3 to 19. Table 2 will also be printed on the terminal.
We recommend removing these figures before moving forward, especially if one aims to reproduce all the security results from scratch. Use the following command:

```
$ rm *.pdf
```

A.4 Evaluation workflow

We provide flexibility in how one can reproduce our security results. A single script can be used to run experiments for all security figures (Figures 3–19) at once. We also provide the option to use our existing simulation results to generate the figures without a fresh run:

```
$ bash genAllFigs.sh 1
```

In order to generate all figures using a fresh simulations, run:

```
$ bash genAllFigs.sh 0
```

Similarly, to generate Table-2 using existing results, run:

```
$ bash genTable.sh 1
```

A fresh set of simulations for Table-2 can be run using:

```
$ bash genTable.sh 0
```

All scripts that will be discussed later have the same option of using either our existing results (set option to `1`) or running a fresh set of simulations (set option to `0`). We assume that the user wants a fresh set of simulations, and set this option to `0` by default in Section A.4.2.

A.4.1 Major Claims

(C1): Increasing the number of skews helps reduce the eviction rate. Additionally, load-aware skew selection improves eviction rate. These are validated by **E1** described in Section 3.2, with results in Figures 3 and 4. We later talk about how this depends on the warm-up state in **Ex**.

(C2): Only the combination of skews with load-aware skew selection, invalid tags and global LRU eviction provides a security benefit. Additionally, increasing the invalid tags improves the security of such cache designs. These are validated by **E2** described in Section 3.3, with results in Figures 5, 6 and 7.

(C3): High associativity provides significant security benefits. This is validated by **E3** described in Section 3.4, with results in Figures 8 and 9.

(C4): Random replacement policy performs worse than LRU and RRIP for conflict-based attacks, as it results in higher eviction rates. This is validated by **E4** described in Section 3.5, with results in Figures 10 and 11.

(C5): High associativity designs require significantly more LLC evictions to create eviction sets that achieve a 30% eviction rate. Adding the knobs of load-aware skew selection, invalid tags, and global eviction further improves the security. These are validated by **E5** described in Section 3.6, with results in Figure 12.

(C6): The conflict testing algorithm is much faster than Prime, Prune and Probe in terms of number of LLC evictions needed for same-sized eviction sets. This is validated by **E6** described in Section 3.6, with results in Table 2.

(C7): Eviction rate is independent of the cache size (assuming same associativity). However, the number of LLC evictions required to create eviction sets that achieve a 30% eviction rate increase with cache size. These are validated by **E7** described in Section 3.7, with results in Figures 13 and 14.

(C8): SassCache and way-based partitioned cache designs perform significantly better than other randomized cache designs against occupancy-based attacks. Also, a random replacement policy is better suited against occupancy-based attacks compared to a deterministic replacement policy. These are validated by **E8** described in Section 4, with results in Figures 15 and 16.

(C9): Mirage is vulnerable to low-occupancy-based attacks compared to other randomized cache designs discussed in this work. This is validated by **E9** described in Section 4, with results in Figure 17.

(C10): The security benefit of designs using load-aware skew selection has a strong dependence on the cache warm-up state. This is validated by **E10** described in Appendix B, with results in Figures 18 and 19.

A.4.2 Experiments

(E1): [Impact of skews and skew selection policy] [48 compute-hour]: This experiment analyzes the impact of skews and the skew-selection policy on cache security.
Execution: To run the experiment:

```
$ cd cache-model/
$ python3 get-figure.py 0 3
$ python3 get-figure.py 0 4
```

Results: `figure3.pdf` and `figure4.pdf` files are generated.

(E2): [Combinations of skew selection policies, invalid tags, and eviction policy] [72 compute-hour]: This experiment analyzes the security of various knob combinations including skews with random and load-aware skew selection, invalid tags, and local and global eviction.
Execution: To run the experiment:

```
$ cd cache-model/
$ python3 get-figure.py 0 5
$ python3 get-figure.py 0 6
$ python3 get-figure.py 0 7
```

Results: `figure5.pdf`, `figure6.pdf` and `figure7.pdf` files are generated.

(E3): [Impact of high associativity] [48 compute-hour]: This experiment analyzes the security impact of high associativity on designs with just two skews, and also invalid-way-based designs.
Execution: To run the experiment:

```
$ cd cache-model/
$ python3 get-figure.py 0 8
$ python3 get-figure.py 0 9
```

Results: `figure8.pdf` and `figure9.pdf` files are generated.

(E4): [Impact of replacement policy] [48 compute-hour]: This experiment analyzes the security impact of the replacement policy on designs with just two skews and high associativity, and also invalid-way-based designs.
Execution: To run the experiment:

```
$ cd cache-model/
$ python3 get-figure.py 0 10
$ python3 get-figure.py 0 11
```

Results: `figure10.pdf` and `figure11.pdf` files are generated.

(E5): [Number of LLC evictions required for eviction-set-creation] [96 compute-hour]: This experiment evaluates the number of LLC evictions, required to create an eviction set achieving 30% eviction rate, on various designs.
Execution: To run the experiment:

```
$ cd cache-model/
$ python3 get-figure.py 0 12
```

Results: `figure12.pdf` file is generated.

(E6): [Comparison of eviction-set-creation policies] [96 compute-hour]: This experiment compares the conflict testing and prime, prune and probe algorithms based on the number of LLC evictions they require to create same-sized eviction sets.
Execution: To run the experiment:

```
$ bash genTable.sh 0
```

Results: Table-2 is printed on the terminal.

(E7): [Sensitivity to cache size] [96 compute-hour]: This experiment analyzes the impact of cache size on the eviction rate as well as on the number of LLC evictions required for eviction-set-creation.
Execution: To run the experiment:

```
$ cd cache-model/
$ python3 get-figure.py 0 13
$ python3 get-figure.py 0 14
```

Results: `figure13.pdf` and `figure14.pdf` files are generated.

(E8): [Security against occupancy-based attacks] [96 compute-hour]: This experiment analyzes the security of various cache designs against occupancy-based attacks.
Execution: To run the experiment:

```
$ cd cachefx/
$ python3 get-figure.py 0 15
$ python3 get-figure.py 0 16
```

Results: `figure15.pdf` and `figure16.pdf` files are generated.

(E9): [Security against low-occupancy-based attacks] [30 human-minute + 128 compute-hour]: This experiment analyzes the security of various cache designs against low-occupancy-based attacks.
Execution: To run the experiment:

```
$ cd low-occupancy/
$ sudo docker run -it -v $(pwd)/randomized
_caches:/home/randomized_caches
randomized-caches
$ cd randomized_cache_hello_world/
$ bash setup.sh
$ cd ../ ; bash buildAES.sh
$ bash genNumbers.sh
$ bash getGE.sh
$ exit
$ sudo mv randomized_caches/results results
$ python3 get-figure.py 0
```

Results: `figure17.pdf` file is generated.

(E10): [Impact of warm-up states] [48 compute-hour]: This experiment analyzes the impact of warm-up state on the eviction rate, and how it causes deviation in results from the original eviction-rate experiment.

Execution: To run the experiment:

```
cd cache-model/
python3 get-figure.py 0 18
python3 get-figure.py 0 19
```

Results: `figure18.pdf` and `figure19.pdf` files are generated.

A.5 Notes on Reusability

We highly encourage others to use this work in order to analyze other cache designs and configurations. For `cache-model` simulations, one can change the cache configuration in `cache-model/config/cache.json`. If needed, the source code inside `cache-model/cache` can be modified to implement additional cache designs. New experiments can be implemented by modifying the source code in `cache-model/test`. For `cachefx` simulations, new configurations can be added in `cachefx/configs` and new cache designs can be implemented using the files in `cachefx/Cache`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: TEEcorrelate: An Information-Preserving Defense against Performance-Counter Attacks on TEEs

Hannes Weissteiner[1], Fabian Rauscher[1], Robin Leander Schröder[3,4], Jonas Juffinger[1]
Stefan Gast[1], Jan Wichelmann[2], Thomas Eisenbarth[2], Daniel Gruss[1]
[1]Graz University of Technology, Austria, [2]University Luebeck
[3]Fraunhofer SIT, Darmstadt, Germany, [4]Fraunhofer Austria, Vienna

A Artifact Appendix

A.1 Abstract

Trusted-execution environments (TEEs) offer confidentiality in shared environments. While Intel restricts performance counter access, limiting load-balancing and anomaly detection on TEEs, AMD exposes performance counters to the host, leaving the TEE vulnerable to side-channel leakage. In this paper, we propose TEEcorrelate, a lightweight information-preserving defense against performance counter attacks on TEEs. TEEcorrelate reconciles monitoring capabilities of the host and confidentiality requirements of the TEE, by statistically decorrelating performance counters. TEEcorrelate combines two components, temporal decorrelation using counter aggregation windows, and value decorrelation using fuzzy performance counter increases.

The artifact for this paper models the behavior of TEEcorrelate to analyze sample sizes, probabilities, and trend-following performance. Releasing the artifact allows for simulation and tweaking of the defense.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact does not perform any destructive or privacy-invasive actions.

A.2.2 How to access

Our artifact is available on Zenodo at https://doi.org/10.5281/zenodo.15699920.

A.2.3 Hardware dependencies

As our artifact only simulates the behavior of TEEcorrelate, it does not require any specific hardware. To generate sample data, we require a system that supports performance counters, which are available on most modern CPUs.

A.2.4 Software dependencies

Our artifact requires Python 3 (tested on Python 3.12.10), and the following Python packages:

- numpy
- matplotlib
- scipy

To collect sample data, we use the Linux `perf` tool, which is available on most Linux distributions.

A.2.5 Benchmarks

None.

A.3 Set-up

Set-up requires installing Python 3 along with the required packages. On Ubuntu, this can be done with the following command:

```
sudo apt install python3 python3-scipy \
python3-numpy python3-matplotlib
```

If the Nix package manager is available, it is possible to create a shell with all required dependencies by running:

```
nix-shell -p python3 python3.pkgs.matplotlib \
python3.pkgs.scipy python3.pkgs.numpy
```

A.3.1 Installation

Download the artifact from Zenodo at https://doi.org/10.5281/zenodo.15699920 and extract the archive.

A.3.2 Basic Test

The artifact contains multiple directories, each containing different experiment scripts. Each script can be run independently without any additional parameters, except for two: `base_simulation.py` is a base class for the other scripts

which does not run on its own, and `simulate_defense.py` requires a parameter to specify the path to a trace file. We have included a sample trace file `sample_trace.txt` in the `simulate_defense` directory. Run the different scripts (`python <script_name>`) to perform the experiments.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): TEEcorrelate decorrelates the reported signal from the real signal, while still following coarse-grained trends. This is demonstrated by the experiments (E1) and (E2).

(C2): TEEcorrelate's fast normal distribution approximates a low-resolution binomial distribution. This is demonstrated by experiment (E3).

(C3): The number of required samples to break TEEcorrelate's defense is high, and not significantly affected by using a fast normal distribution instead of a binomial distribution. This is demonstrated by experiment (E4) and (E5).

A.4.2 Experiments

(E1): Minimal Simulation [3 human-minutes]:
How to: Run the minimal defense simulation script `defense_simulation/simulate_small_trace.py`.
Results: A figure gets rendered, showing the original values (blue), and the decorrelated values (orange) for a small, simulated trace. The green area signifies the maximum deviation. Cf. Figure 4 in the paper.

(E2): Full Simulation [10 human-minutes]:
How to: Run the full defense simulation script `defense_simulation/simulate_defense.py` with a trace file (e.g., the included `sample_trace.txt`).
Optional: [20 compute-minutes] A custom trace file can be generated using the Linux `perf` tool, e.g., by running the following command:

```
perf stat -e ex_ret_instr -I 1 \
-e l2_pf_miss_l2_l3 > trace.txt
```

This command runs until interrupted, and collects performance counters every millisecond. It is important that one of the counters counts the number of retired instructions, and that the name of the counter is edited in line 46 when not using the AMD `ex_ret_instr` counter. Target performance counter names need to be added to the `perfcounts` list in the script.
Results: The left figure shows the original values (blue), and the decorrelated values (orange) for a real performance counter trace. On the right, we display the distribution of the offset between real and decorrelated values (cf. Figure 5 in the paper). Mean and standard deviation of the offset are printed to the console.

(E3): Distribution Simulation [5 human-minutes + 10 compute-minutes]:
How to: Run the distribution simulation script `distribution_samplesizes/generate_distribution_shape.py`. The script runs until interrupted, and updates the figure every 200000 samples.
Results: The figure shows a binomial-like distribution. Thus, as claimed in (C2), the fast normal distribution approximates a low-resolution (64 samples) binomial distribution, while having a much larger output parameter space.

(E4): Sample Size Calculation [15 human-minutes]:
How to: Run the sample size calculation script `distribution_samplesizes/distribution_with_32bit.py`. The script generates a figure by calculating the probability gradient (i.e., the difference in probability between each possible offset and the next) for a 32-bit fast normal distribution, a ideal gaussian distribution, and 64-bit distributions with multiple offset values.
Optional: The default parameter of this script is set to a deviation window size of 2048. To compute other window sizes, the parameter `DEVIATION_RANGE` in the script can be changed to the desired value.
Results: Figure 1 displays the CDF for the 3 different functions (normal, 64-bit fast normal, and 32-bit fast normal). Figure 2 displays the probability gradient for the same functions. Figure 3 displays the number of required samples to distinguish two probabilities, at each point of the curves. Figures 1, 2, and 3 generated by this script correspond to figures 8, 6, and 10 in the paper, respectively. We also print the minimum number of required samples for each parameter set, which can be compared to table 3 in the paper.

(E5): HQC Attack Runtime Computation [5 human-minutes, 5 compute-minutes]:
How to: Run the HQC runtime estimation script `distribution_samplesizes/hqc_runtime.py`. The script generates approximate attack runtimes for an attack on HQC with TEEcorrelate enabled, with for the deviation window sizes of 64, 2048, and 32768.
Results: For each window size and desired oracle accuracy, we print the estimated runtime (in seconds) for a single oracle query, the total runtime, and the number of required *fast divisions* to the console. the minimum calculated runtime can be compared with our claims at the end of section 6.4 in the paper.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Systematic Evaluation of Randomized Cache Designs against Cache Occupancy

Anirban Chakraborty*† Nimish Mishra*‡ Sayandeep Saha§
Sarani Bhattacharya‡ Debdeep Mukhopadhyay‡

†Max Planck Institute for Security and Privacy, Germany
‡Indian Institute of Technology Kharagpur, India
§Indian Institute of Technology Bombay, India

anirban.chakraborty@mpi-sp.org nimish.mishra@kgpian.iitkgp.ac.in {sarani, debdeep}@cse.iitkgp.ac.in sayandeepsaha@cse.iitb.ac.in

A Artifact Appendix

A.1 Abstract

This work performs a systematic evaluation of 5 randomized cache designs- CEASER, CEASER-S, MIRAGE, ScatterCache, and SassCache against cache occupancy wrt. *both* performance as well as security. This work fills in a crucial gap in current literature on randomized caches: currently most randomized cache designs defend only contention-based attacks, and leave out considerations of cache occupancy. With respect to performance, this work proposes a new and uniform benchmarking strategy, which allows us to perform a fair and comparative analysis across all designs under various replacement policies. Likewise, with respect to security against cache occupancy attacks, this work evaluate the cache designs against various threat assumptions: ① covert channels, ② process fingerprinting, and ③ AES key recovery (to the best of our knowledge, this work is the first to demonstrate full AES key recovery on a randomized cache design using cache occupancy attack). The main takeaway of our work is to establish the need to *also* consider cache occupancy side-channel in randomized cache design considerations.

In this artifact appendix, we detail steps to reproduce our results from the paper. We first detail how to build and run the different randomized cache designs in gem5. MIRAGE and ScatterCache are already open-sourced, while CEASER and CEASER-S have been implemented in-house by us and open-sourced. The implementation of SassCache has been generously contributed by SassCache's authors and adapted to our setup. Thereafter, we detail steps to reproduce the various experiments related to performance and security analysis detailed in the paper.

*Equal Contribution.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact does not have any destructive tendencies, and is thus safe to use. Moreover, we do not attack any commercial library/implementation in this work, and thereby do not come under the purview of disclosure and release of security advisories.

A.2.2 How to access

The permanent link for the source codes used in this paper can be found at: `https://github.com/SEAL-IIT-KGP/randomized_caches/tree/main` (tag: v3.0). The artifact is also hosted at Zenodo; url: `https://doi.org/10.5281/zenodo.15198048`.

A.2.3 Hardware dependencies

We do not have any specific requirements for hardware. However, it is recommended to have at least 16 MB of RAM for smooth gem5 simulations. For all our experiments reported in this appendix, we used Intel(R) Xeon(R) Gold 6226*R* CPU as the evaluation platform (running Ubuntu 20.04.6 LTS, 32 GB RAM).

A.2.4 Software dependencies

We have the following major requirements: `gcc-9`, `g++-9`, `python2.7`, and `virtualenv`. ScatterCache specifically requires `gcc-7` and `g++-7`. Other dependencies include: `git`, `python2.7-dev`, `wget`, `curl`, `pip`, `zlib`, `HDF5`, `M4 macroprocessor`, `matplotlib`, and `python-dev`. Rest dependencies are installed automatically by the setup scripts (detailed below).

A.2.5 Benchmarks

Experiments related to performance benchmarking and fingerprinting attack require SPEC2017 should our results be reproduced from scratch. Alternatively, the results can also be derived from the in-house datasets related to these experiments that we open-source. We do not provide SPEC2017 sources as part of the artifact due to license constraints.

A.3 Set-up

A.3.1 Installation

Navigate to the subdirectory `randomized_cache_hello_world/` and execute `setup.sh`. This script sets up a virtual environment in Python 2.7, installs the needed dependencies (like `scons` and `six`), and builds the gem5 binaries of all cache designs.

The bottleneck step in this installation is the number of threads used to build gem5. The parameter `THREADS` in `setup.sh` controls the amount of parallelization: more threads imply shorter build time, but also uses more RAM. We have set `THREADS` to a conservative 1, but recommend tweaking it based on the system configuration.

A.3.2 Basic Test

The subdirectory `randomized_cache_hello_world/` contains scripts: `run_ceaser.sh`, `run_ceaser_s.sh`, `run_mirage.sh`, `run_scatter.sh`, and `run_sass.sh`. Each of these scripts build a sample binary (i.e. `spurious_occupancy.c`) and executes the same within the gem5 simulation of its respective cache architecture. A successful simulation is implied by an exit message of form:

***Spurious Occupancy step finished**. Exiting @ tick 67659457149 because exiting with last active thread context*

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Performance Evaluation: One of the main claims of the paper is that prior benchmarking techniques are unsuitable for a fair evaluation of modern randomized cache designs. Section 3.3 introduces the idea of benchmarking under spurious occupancy. We claim this benchmarking strategy provides a fair evaluation for all cache designs by (1) removing any implementation specific assumption, and (2) harmonizing the parameters considered for benchmarking different designs.

(C2): Performance Evaluation under different Replacement Policies: This work also shows the effect of replacement policies on the performance of different cache designs, allowing for a better understanding and takeaways for the performance different cache designs. This claim pertains to Section 3.4 in the main paper.

(C3): Covert Channel: An important claim of the paper is that certain kinds of randomized cache designs alleviate creation of covert channels. We claim that design rationale based on pseudo-fully associative design are more amenable to creation of covert channels than set-associative designs. This claim pertains to Section 5 in the main paper.

(C4): Process Fingerprinting: We claim that pseudo-fully associative design are more amenable to process fingerprinting side-channel attacks than set-associative designs. This claim pertains to Section 6 in the main paper.

(C5): AES key recovery: We claim that pseudo-fully associative design are more amenable to AES key recovery side-channel attacks than set-associative designs. This claim pertains to Section 7 in the main paper.

A.4.2 Experiments

(E1): *[Performance Evaluation] [10 human-minutes + 24 compute-hours + 3GB disk]:* To reproduce results related to claim **C1**.

How to: Navigate to subdirectory `perf_runs`. It has the scripts *run_baseline_benchmark.sh*, *run_ceaser_benchmark.sh*, *run_ceaser_s_benchmark.sh*, *run_mirage_benchmark.sh*, *run_sass_benchmark.sh*, and *run_scatter_benchmark.sh* that help run SPEC2017 benchmarks on the associated cache designs.

Preparation: The environment variable `BASE_DIR` must point to the top-level directory of the artifact, while the environment variable `SPEC_PATH` points to the SPEC2017 installation.

Execution: Run one of the scripts, providing a benchmark name as input. The list of supported benchmarks is captured in the README file of the `perf_runs` subdirectory. For example, *bash run_mirage_benchmark.sh blender* runs blender on MIRAGE.

Results: The collected measurements must be put into `perf_runs/perf_data/randomized_caches.xlsx` to reproduce the results in Section 3.3.

Data: Alternatively, we also provide our dataset in `perf_runs/perf_data/`. The sub-directory names are a combination of the cache design and the replacement policies. To reproduce results related to claim **C1**, consider only the subdirectories with suffix `RandomRP`.

Each subdirectory contains further directories for each benchmark. For instance, `ceaser_perf_runs_RandomRP/gcc` contains the `stats.txt` for running the SPEC2017 benchmark gcc on CEASER under the default replacement pol-

icy. A helper script `compile_stats.sh` will aid in constructing the requisite data from these sub-directories

We have already abstracted out results of `compile_stats.sh` for each subdirectory of `perf_runs/perf_data/` into `perf_runs/perf_data/randomized_caches.xlsx`. This spreadsheet contains formulae for automatically computing the LLC miss ratio, normalize it against the baseline, and perform performance evaluation. It reproduces Table 2 of the paper, as well as computes data used by the scripts in `$BASE_DIR/scripts` which plot Figure 5 in the main paper..

(E2): *[Performance Evaluation under Replacement Policies] [10 human-minutes + 24 compute-hour + 3GB disk]:* To reproduce results related to claim **C2**.

How to: Same as **E1**.

Preparation: Same as **E1**.

Execution: Same as **E1**, except that the replacement policy can also be provided as input. Refer to the README in the artifact for more details.

Results: Same as **E1**

Data: Same as **E1**. This experiment reproduces Table 2 of the paper.

(E3): *[Covert Channel] [10 human-minutes + 8 compute-hours]:* To reproduce results related to claim **C3**.

How to: Navigate to sub-directory `llc_simulator`.

Preparation: None. This experiment does not require any additional setup. The source codes are mostly written in `python3` and uses basic packages such as `numpy` and `matplotlib`.

Execution: Execute `bash run_simulation.sh`. This script executes occupancy based covert channel setup for MIRAGE, CEASER, ScatterCache, and Baseline. The scripts collect covert channel data and write to `outfile_*.txt` for both bits 0 and 1. Finally, `covert_channel_plot.py` plots the collected data.

Results: This experiment validates the claim **C3**. Overall, this experiment establishes that randomized caches that rely on pseudo-fully associative design rationale are susceptible to covert channel setup (with high accuracy) even with 10% accuracy. On the other hand, comparatively, it is difficult to setup a covert channel of acceptable accuracy with 10% accuracy.

(E4): *[Process Fingerprinting] [10 human-minutes + 48 compute-hours + 8GB disk]:* To reproduce results related to claim **C4**.

How to: Navigate to sub-directory `fingerprinting`.

Preparation: Same as **E1** and **E2** since this experiment requires setting up SPEC2017.

Execution: Follow the steps README illustrates in the subdirectory. Essentially, this experiment is similar to **E1**, except that the benchmarks are randomly selected.

Results: Use the scripts `ceaser.py`, `ceaser_s.py`, `mirage.py`, `sass.py`, and `scatter.py` to construct accuracy metrics for the fingerprinting experiment.

Data: We have open-sourced the data collected internally in `fingerprinting/data/` as `*.log` files. Running the scripts `ceaser.py`, `ceaser_s.py`, `mirage.py`, `sass.py`, and `scatter.py` on these log files creates the data used to construct Figure 8 in the main paper (and the inferences drawn in Section 6).

(E5): *[AES] [10 human-minutes + 1 compute-hour + 0.5 GB disk]:* To reproduce results related to claim **C5**.

How to: Navigate to the directory `aes`. It has two subdirectories: `aes_profiled_key` and `aes_victim_key`. The C implementations of the AES T-Box within these subdirectories contain the attack and victim code. `aes_profiled_key` contains the key being used for profiling: **0xffeeddccbbaa99887766554433221100**. On the other hand, `aes_victim_key` contains the key being recovered from the constructed profile: **0x7766554433221100ffeeddccbbaa9988**. All data is collected at 50% occupancy (Section 7.3 and 7.4) and suffices to establish the reproducibility of our attack.

Preparation: None

Execution: The `analysis` subdirectory contains the dataset for each randomized cache design. For each cache design, we provide a script `analysis.py` that constructs the profile and mounts the attack (as per Section 7.1 of the paper), as well as a script `guessing_entropy.py` that computes the guessing entropy.

Results: The computed guessing entropy for the cache design comes out as: MIRAGE : 33.0, CEASER-S : 92.0, SassCache : 104.0, and ScatterCache : 100.0. Note that the results in Section 7.3 have been compiled over runs over multiple different keys while the artifact contains representative runs of a single key. This implies the exact value of the guessing entropy in the paper is different that what these scripts output. However, the *general trend* of strength/weakness of each design against AES key recovery attacks still stands. More precisely:

1. GE for MIRAGE: Paper [30.05], Representative data in artifact [33]
2. GE for SassCache: Paper [109.31], Representative data in artifact [104]
3. GE for ScatterCache: Paper [100.97], Representative data in artifact [100]

4. GE for CEASER-S: Paper [90.56], Representative data in artifact [92.0]

As evident, MIRAGE is the weakest with GE . Then comes CEASER-S with a GE of ; key recovery is not possible with this GE, but CEASER-S does show leakage. Then we have ScatterCache with a GE of 3, demonstrating *some* leakage, but not exploitable. Finally, SassCache is the most resilient against our attacks.

AES data collection from scratch: We encourage the user to try out data collection with different keys to get a trend (and GE) closer to what is reported in the paper. However, as also noted in the paper (footnote 21), the rate of data collection is at best 500 observations per hour. We were able to thereby deploy about 350 cores per Intel Xeon server, across three such servers. The overall data collection for *all* designs and multiple keys took over 2 weeks of compute hours. We thus do not include any data collection experiment for AES in artifact evaluation because of the abnormally high runtime of completing the experiment.

Note that such an inhibitory rate is *not a problem of attack design*, but rather is the consequence of the gem5 simulations (which is the go-to simulation strategy for state-of-the-art randomized cache literature). In a realistic setting (when these randomized caches are deployed on real hardware), our attack will be much faster.

A.4.3 Plot Scripts

We also provide the scripts in `scripts/` that we used to generate the plots in the paper.

A.5 Notes on Reusability

The findings of this thereby establish an interesting open problem: Design of a randomized cache of comparable efficiency with modern set-associative LLCs, while still resisting *both* contention-based and occupancy-based attacks. We hypothesize that randomized caches designed around dynamically changing partitions would be capable of restricting LLC occupancy for critical workloads (thereby providing security), while also generously allowing LLC occupancy for non-critical workloads (thereby providing better performance in general). More research however is required to concretely establish how such a cache design shall function, how process requests for dynamic changes in partitioning will be handled, and how to implement this design in an uncomplicated manner. For such future research, our artifacts will enable designers to test upcoming designs against occupancy attacks as well.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

A.7 Acknowledgements

We would like to thank the authors of SassCache who very generously provided us with their custom simulator for SassCache upon request, which we have adapted as part of this artifact.



USENIX Security '25 Artifact Appendix: Exploiting Inaccurate Branch History in Side-Channel Attacks

Yuhui Zhu[1,2] and Alessandro Biondi[1]

[1]*Scuola Superiore Sant'Anna*
[2]*Scuola IMT Alti Studi Lucca*

A Artifact Appendix

A.1 Abstract

Modern out-of-order CPUs heavily rely on speculative execution for performance optimization, with branch prediction serving as a cornerstone to minimize pipeline stalls and maximize efficiency. When shared branch prediction resources lack proper isolation and sanitization methods, they can introduce security vulnerabilities that expose sensitive data across different software contexts.

This artifact evaluates the behavior of two underdocumented features of the Branch Predictor Unit: *Bias-Free Branch Prediction* and *Branch History Speculation*. These discoveries expose previously unknown cross-privilege attack surfaces for Branch History Injection (BHI).

Based on these findings, we present three novel attack primitives: two Spectre attacks, namely **Spectre-BSE** and **Spectre-BHS**, and a cross-privilege control flow side-channel attack called **BiasScope**. This artifact evaluates the presence of these primitives using user-mode intra-process proof-of-concepts, then evaluates their capability for mounting cross-privilege attacks using custom syscall handlers. Finally, we demonstrate the **Chimera** snippet using eBPF to achieve end-to-end exploitation.

A.2 Description & Requirements

This artifact contains proof-of-concept (PoC) code demonstrating the vulnerabilities discovered in the paper. The project is organized into several submodules, each addressing different attack scenarios:

- **`intra-ctx/`:** Intra-process PoCs demonstrating the relevant microarchitectural behaviors and primitives to perturb and exploit their side effects. This module covers BHB/PHT mistraining (Section 3.3), Spectre-BSE (Section 5.4), Spectre-BHS (Section 6.2), and Chimera snippets (Section 7).

- **`cross-ctx/`:** Cross-context PoCs showcasing how t

  Intra-context demos. The `intra-ctx/` module consists of multiple sub-modules (implemented by `intra-ctx/tests/*.c`) which demonstrate the exploited primitives, the corresponding microarchitectural behaviors, and basic attack flows described in the paper using intra-mode attack processes.

  Cross-context demos. hese primitives can manipulate branch prediction in kernel mode or another process. This module covers BiasScope (Section 5.3), Spectre-BSE (Section 5.4), and Spectre-BHS (Section 6.2).

- **`chimera-ebpf/`:** End-to-end Chimera attack (Section 7) implemented as an eBPF program, demonstrating practical kernel memory leakage.

Please refer to the `README.md` files in the root directory and each subdirectory for details of all these submodules.

A.2.1 Security, privacy, and ethical concerns

While the artifact does not collect, transmit, or store any personal data and poses minimal privacy risks, evaluators should be aware of several important risks. The cross-context demonstrations require patching and recompiling the Linux kernel on ARM machines, which may affect system stability and integrity, so evaluators should use dedicated test machines rather than production systems. Additionally, cross-privilege BiasScope evaluation requires disabling Spectre mitigations via `mitigations=off`, which temporarily reduces the system's protection against speculative execution attacks and should be reverted after evaluation. After evaluation, the modified kernel and disabled mitigations need to be carefully reset to restore the original security posture.

A.2.2 How to access

The artifact source code is publicly available on Zenodo at `https://zenodo.org/records/15612187`. The archived repository includes complete documentation with

detailed setup and execution instructions. Reviewers may evaluate these artifacts on evaluator-owned machines, provided their processors match the vulnerable models documented in our paper. Please follow the instructions provided in all `README.md` files.

A.2.3 Hardware dependencies

This artifact requires specific processor architectures to demonstrate the novel branch prediction vulnerabilities described in our paper. To fully reproduce the findings, evaluators will need access to one or more of the following processor families listed in Table 1.

A.2.4 Software dependencies

To compile the intra-context module of the artifact, `make` and `gcc` are required. For cross-compilation targeting ARM machines, you will need the `aarch64-linux-gnu-gcc` toolchain installed on your system.

Compiling or cross-compiling the patched kernel requires the standard kernel build toolchain, including appropriate cross-compilation tools for target architectures.

The artifact is compatible with any standard Linux distribution and does not require specific distribution versions.

For intra-context demonstrations in the `intra-ctx/` directory, reviewers can simply cross-compile the PoCs for their target architecture and execute them with provided parameters in standard userspace without kernel modifications.

Cross-context demonstrations in the `cross-ctx/` directory require more advanced setup including applying provided kernel patches to inject victim code into kernel space, recompiling and installing a modified kernel. Additionally, cross-privilege BiasScope demonstrations require disabling Spectre mitigations via bootloader (`mitigations=off`).

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Compiling Intra-Context PoCs

The intra-context demonstrations in `intra-ctx/` require no special setup in the evaluation environment. These self-contained programs can be compiled and executed in standard userspace. To build a test, first clean any existing builds:

```
make clean
```

For native compilation on the target architecture (e.g., building on an x86 machine for itself), use:

```
make TEST=<test_name> ARCH=amd64
```

For cross-compilation, specify the appropriate toolchain using `CROSS_COMPILE=`:

```
make TEST=<test_name> ARCH=aarch64 CROSS_COMPILE=
    aarch64-linux-gnu-
```

Multiple intra-context demonstrations are available in the `intra-ctx/tests/` directory, with executables generated as `build/main`:

- `pht-idx`: BTB/PHT mistraining experiments (Sec. 3.3)
- `spec-bse`: Spectre-BSE attack demonstration (Sec. 5.4)
- `spec-bhs`: Spectre-BHS attack demonstration (Sec. 6.2)
- `chimera`: Spectre-BHS attack demonstration using Chimera gadget (Sec. 7)

Some modules support additional debugging flags that can be passed during compilation by appending `FLAGS="FLAG1 FLAG2 ..."` to the make command:

```
make TEST=spec-bhs ARCH=aarch64 CROSS_COMPILE=
    aarch64-linux-gnu- FLAGS="DBG_ARCH_BH
    DBG_JMP_LATENCY"
```

A.3.2 Compiling Cross-Context User-Space Programs

To compile individual cross-context demonstrations in `cross-ctx/`, use the following command:

```
make $DEMO_NAME FLAGS="$FLAGS" TARGET="$TARGET"
```

The compilation requires the following parameters:

- `DEMO_NAME`: Specifies which demonstration to compile. Available options include:
  - `scope-reader`, `scope-writer`: Complementary user-space programs for the BiasScope attack that demonstrate reading from and writing to the Branch Status Table (BST) to establish a covert side channel.
  - `spec-bse-demo-el1`: Cross-privilege demonstration of the Spectre-BSE attack, exploiting Branch Status Eviction vulnerabilities.
  - `spec-bhs-demo-el1`: Cross-privilege demonstration of the Spectre-BHS attack, leveraging Branch History Speculation mechanisms.
- `TARGET`: Defines the target platform for compilation, selecting appropriate platform-specific parameters from `target.h`. Currently supported targets are `imx8` (i.MX8QuadMax MEK) and `rpi5` (Raspberry Pi 5).
- `FLAGS`: Optional compilation flags that enable specific debugging features or attack variants. Platform-specific flags are detailed in the respective demonstration sections below.

μarch	Attack Compatibility	Validated SoC
ARM Processors		
Cortex-A72 (<= r0p3)	BiasScope, Spectre-BSE	NXP i.MX8QuadMax MEK
Cortex-A76	Spectre-BHS, Chimera	Raspberry Pi 5
Cortex-A78/A78AE	Spectre-BHS, Chimera	NVIDIA Jetson AGX Orin
x86 Processors		
AMD Zen4	Spectre-BHS (only 'intra-ctx/'), Chimera	Ryzen 7 7840U
Intel Gracemont	Spectre-BHS (only 'intra-ctx/'), Chimera	N100
Intel Redwood Cove/Crestmont	Spectre-BHS (only 'intra-ctx/'), Chimera	Core Ultra 7 155H

Table 1: Availability of demos on tested hardware platforms.

A.3.3 Setting Up Custom Syscall Handlers

Cross-context demonstrations require custom syscall handlers on ARM processors to facilitate kernel-space attack components. You must patch the kernel using the code provided in `cross-ctx/kernel/` to establish these kernel-space components.

1. Download the Linux kernel source code for your target platform:
   - For Cortex-A76 (Raspberry Pi 5): Use the Linux kernel `rpi-6.6.y` branch from the Raspberry Pi kernel repository (https://github.com/raspberrypi/linux/tree/rpi-6.6.y).
   - For Cortex-A72 (i.MX8QuadMax MEK): Use the kernel included in the Yocto project distributed by NXP, version `imx-5.15.71-2.2.2` (https://github.com/nxp-imx/imx-manifest/blob/imx-linux-kirkstone/imx-5.15.71-2.2.2.xml).
2. Copy the custom syscall handler files from `custom_syscall/` to the kernel source code directory.
3. Apply the appropriate patch to `include/uapi/asm-generic/unistd.h` to register the syscall numbers:
   - For Raspberry Pi 5: Apply `rpi-6.6.y.diff`
   - For i.MX8: Apply `lf-5.15.71-2.2.2.diff`
4. Compile and install the modified kernel with the custom syscall handlers:
   - For Raspberry Pi 5: Follow the Raspberry Pi kernel documentation (https://www.raspberrypi.com/documentation/computers/linux_kernel.html)
   - For i.MX8: Follow the NXP community guidance (https://community.nxp.com/t5/i-MX-Processors-Knowledge-Base/i-MX-Yocto-Project-How-can-I-quickly-modify-the-kernel-and-test/ta-p/1129551)

A.3.4 Chimera eBPF PoC

To build the Chimera eBPF module in `chimera-ebpf/`, use one of the following commands based on your compilation target and platform:

```
make CROSS_COMPILE=aarch64-linux-gnu- ARCH=aarch64
make ARCH=amd64
```

A.3.5 Installation

This artifact does not require any installation process and is ready to run immediately after compilation. For installing the patched kernel required by cross-context demonstrations, please refer to the platform-specific documentation provided by your SoC or board manufacturer, as detailed in the kernel setup instructions above.

A.3.6 Basic Test

Run the intra-context `pht-idx` demonstration to test whether you can successfully manipulate branch prediction as described in Section 3.3 in the paper. Please follow the instructions in `intra-ctx/README.md` for guidance and expected results.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): On Cortex-A72, the presence of a Branch Status Table yields two potential side-channel attack primitives, Spectre-BSE and BiasScope. This is demonstrated by experiments (E1) and (E3).

(C2): On other processors listed in our paper, we can manipulate the Branch History Speculation mechanism to influence history-based branch prediction and induce unintended mis-speculation through Spectre-BHS primitice. This is demonstrated by experiments (E2), (E3), and (E4).

A.4.2 Experiments

(E1): Passing dummy secrets through BiasScope side channel (1 human-minute + 1 compute-hour + 10GB disk):
How to: Patch the kernel, install the patched kernel, and build the corresponding cross-context demos `scope-reader.c` and `scope-writer.c` in `cross-ctx/`.
Preparation and execution: Follow the instructions in `cross-ctx/README.md` within the code package to build the demo with kernel-user or user-user flavor. Note that to demonstrate the kernel-user flavor of this primitive, you must patch the kernel. Compiling the patched kernel may take longer time and more disk space.
Results: By running the built demo program with parameters described in `cross-ctx/README.md`, you should be able to see the dummy secret being decoded by the `scope-reader.c` program.

(E2): Inducing mis-speculation using Spectre-BSE and Spectre-BHS in the same user process (1 human-minute + 1 compute-minute + 1MB disk):
How to: Build the intra-context demos in `intra-ctx/`.
Preparation and execution: Follow the instructions in `intra-ctx/README.md` within the code package to build your selected demo.
Results: By running the built demo program with parameters described in `intra-ctx/README.md`, you should be able to see microarchitectural marks indicating a successful Spectre-BSE or Spectre-BHS attack.

(E3): Inducing kernel-space mis-speculation using Spectre-BSE and Spectre-BHS from user-space (1 human-minute + 1 compute-hour + 10GB disk):
How to: Patch the kernel, install the patched kernel, and build the cross-context demos in `cross-ctx/`.
Preparation and execution: Follow the instructions in `cross-ctx/README.md` within the code package to build your selected demo. Note that compiling the patched kernel may take longer time and more disk space.
Results: By running the built demo program with parameters described in `cross-ctx/README.md`, you should be able to see microarchitectural marks indicating a successful cross-privilege Spectre-BSE or Spectre-BHS attack.

(E4): Dumping kernel memory with Chimera eBPF PoC (1 human-minute + 1 compute-minute + 1MB disk):
How to: Build the demo in `chimera-ebpf/`.
Preparation and execution: Follow the instructions in `chimera-ebpf/README.md` within the code package to build your selected demo. Note that you may also need to have a known byte sequence in a known kernel address to evaluate the accuracy of the disclosure primitive.
Results: By running the built demo program with parameters described in `chimera-ebpf/README.md`, you should be able to dump a piece of kernel memory at a given address and given length using the side-channel primitive.

A.5 Notes on Reusability

The submodule `intra-ctx/` provides a versatile framework to monitor and evaluate speculative execution. Researchers interested in similar work may implement their own test modules to induce desired mis-speculation using novel techniques. We plan to share updated versions of this code in our git repository in the future.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks

Alvise de Faveri Tron
Vrije Universiteit Amsterdam

Raphael Isemann
Vrije Universiteit Amsterdam

Hany Ragab
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Klaus von Gleissenthall
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

A Artifact Appendix

A.1 Abstract

We present a prototype of our pre-silicon fuzzer that uses a novel detection mechanism based on implicit secrets to find transient data leaks in CPU designs. Our artifact includes all the components of our fuzzing infrastructure, namely (1) an LLVM pass that adds bit-precise taint-tracking to the CPU simulation, (2) an AFL++ fork that adds the option to use taint as feedback, and (3) a LibAFL fuzzing driver, along with our RISC-V instruction generator and mutator, to drive the fuzzing loop. The artifact also includes the detector infrastructure we built for BOOM, namely (1) an architectural simulator that infers all architecturally-accessed locations, (2) the BOOM configuration used for fuzzing and (3) the C++ wrapper that drives the Verilated simulation and detects crashes. The artifact also includes a testsuite of minimal detectable PoCs for both known vulnerabilities (Spectre-v1, Spectre-v2, Meltdown, Spectre-v4, Spectre-RSB) and the new Spectre-LP vulnerability. Information about each component can be found in the README and in the paper.

A.2 Description & Requirements

The scripts provided in the artifact can be used to setup Phantom Trails on the BOOM core and evaluate its results.

A.2.1 Security, privacy, and ethical concerns

Experiments are completely local, and no destructive steps are taken during evaluation.

A.2.2 How to access

The main entrypoint of our artifact is available at `https://github.com/vusec/phantom-trails` at the tag `ae-initial`. The final artifact version will be available on Zenodo at `https://zenodo.org/records/14726711`.

A.2.3 Hardware dependencies

No specific hardware feature is required, however to reproduce the TTE measurements of our fuzzing campaigns it is required to run on comparable machines. Our experiments have been conducted on two machines:

1. (**M1**) AMD Ryzen Threadripper PRO 5995WX machine with 128 cores and 500 GiB of RAM
2. (**M2**) Intel Xeon Silver 4310 machine with 48 cores and 126 GiB of RAM

A.2.4 Software dependencies

No specific OS is required in principle. Our experiments have been tested on Ubuntu Linux 22.04.

A.2.5 Benchmarks

None.

A.3 Set-up

The only prerequisites are `git` and `docker`. Scripts require `bash` and `python3`.

A.3.1 Installation

Clone the repository and all dependencies:

```
git clone https://github.com/vusec/phantom-trails
cd phantom-trails
git submodule update --init --recursive
```

Build the instrumented simulation in a container (takes ≈40 minutes on a 48 cores/126GB RAM machine):

```
cd BOOM
./start.sh
```

By default, this script will build Phantom Trails with basic block coverage as feedback and the BOOM core simulation in its MediumBoom configuration inside of a container named `boom-fuzz-manual-started`. Once the script has terminated, you should get a shell inside of the docker container.

If the build script exits before completion, it is very likely that the process ran out memory during the building and linking phases of LLVM. In that case, you can modify (decrease) the number of parallel jobs by substituting `nproc` at line 21 of `BOOM/start.sh`.

A.3.2 Basic Test

After `start.sh` the shell is located in the `/chipyard/sims/verilator` folder of the Docker container. From there you can run some simple tests.

To verify that the detector works as intended, you can run:

```
phantom-trails run \
    /Samples/build/bins/pocs/meltdown-us.bin
```

This loads the selected RISC-V flat binary in the instrumented simulation's memory and runs it. The source code of the test can be found in `Samples/src/pocs/meltdown-us.S`.

The expected results is that the command will fail, and will indicate "Meltdown PF" as cause before exiting.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Phantom Trails is able to detect all known speculative vulnerabilities on the BOOM core (Spectre-v1, Spectre-v2, Spectre-RSB, Spectre-SSB and Meltdown) at the secret-access step. Experiment **E1** confirms that each PoC of our testsuite is correctly detected and classified, as reported in Section 7.1, paragraph "Poc Detection".

(C2): Phantom Trails was able to uncover a new Spectre-v1 variant, Spectre-LP, described in Section 8 of the paper. Experiment **E2** confirms that different variants of Spectre-LP, including a return-based variant and a branch-based variant, are detected by Phantom Trails.

(C3): Phantom Trails can be extended to handle MDS. Experiment **E3** reproduces the detection behavior as reported in Section 5.4 (MDS Detection).

(C4): Phantom Trails is able to find all known transient vulnerabilities on the BOOM core in reasonable time despite the lack of templates. Experiment **E4** reproduces one of our fuzzing campaigns as described in Section 7.2 (Question Q1), whose results are reported in Figure 8-10 (blue) and in Table 1 (left).

(C5): Phantom Trails is still able to perform similarly to templated approaches on SmallBoomCore. Experiment **E5** reproduces the TTE measurements reported Table 3 (rightmost column).

(C6): Phantom Trails uses simple control-flow and data-flow optimizations. Experiment **E6** described in Section 7.2 (Question Q3) shows the impact of each optimization. Our results are reported in Table 2.

(C7): Phantom Trails provides the possibility of using Taint as a fuzzing feedback. Experiment **E7** described in Section 7.2 (Question Q2) can be used to measure the impact of the taint feedback. Our results are reported in Figure 8-10 (orange) and in Table 1 (right).

A.4.2 Experiments

(E1): PoC Detection [1 compute-minutes]: Run Phantom Trails on PoCs of known vulnerabilities.
Preparation: After running `BOOM/start.sh` you should be running in a `fish` shell inside of the Docker container, in the `/chipyard/sims/verilator` folder.
Execution: Run the following (`fish` syntax)

```
for poc in /Samples/build/bins/pocs/*;
    echo "===== Running $poc ======"
    phantom-trails run $poc;
end;
```

Results: For each sample, Phantom Trails should abort and print "Found issue: " alongside its classification. Samples should include 1 Meltdown sample (reported as `Meltdown_LOAD_PFAULT`), 2 Spectre-RSB samples, 2 Spectre-v1 samples, 1 Spectre-v2 sample, and 2 Spectre-v4 samples.

(E2): Spectre-LP [1 compute-minutes]: Run Phantom Trails on Spectre-LP snippets.
Preparation: You should be running in a `fish` shell inside of the Docker container, in the `/chipyard/sims/verilator` folder.
Execution: Run the following (`fish` syntax)

```
for poc in /Samples/build/bins/spectre-lp/*;
    echo "===== Running $poc ======"
    phantom-trails run --init=/dev/null $poc;
end;
```

Results: The folder includes 3 Spectre-LP PoCs. For each of them, Phantom Trails should abort and print "Found issue: Spectre_v1_new".

(E3): MDS [15 compute-minutes]: Build BOOM with MDS and run Phantom Trails to verify that MDS is detected.
Preparation: Inside of the Docker container, build BOOM with MDS-SB:

```
phantom-trails build --config MDSConfig -j<JOBS>
```

Execution: Run the following

```
phantom-trails run --mds --config=MDSConfig \
    /Samples/build/bins/mds-tests/mds.bin
```

Results: The command is expected to abort after printing "Found issue: MDS_STORE_BUFFER".

(E4): Fuzzing Campaign with SW Feedback [24 to 48 compute-hours]: Run a fuzzing campaign on MediumBoomCore with the Software Feedback.
Preparation: You should be inside of the Docker container. TTEs reported in Table 1 of the paper (left part) have been observed on machine **M1** described in Section A.2.3.
Execution: Run the fuzzer

```
phantom-trails fuzz
```

By default, this will run the fuzzer until all samples in `expected_findings` are found. You can kill the fuzzer at any time with:

```
sudo killall sim-fuzzer && sudo killall run-FuzzConfig
```

Results: While the fuzzer is running, you should see TTEs being updated live in the TUI (top-right panel). Once the fuzzer has found all expected crashes, the results of the campaign are available in `/chipyard/sims/verilator/out`. The same folder is also available outside of the container, under `BOOM/results/manually-started/<timestamp>/`. Crashing inputs after classification can be found in the `out/causes` folder, in flat-binary format. You can retrieve time-to-exposure and iterations-to-exposure from within the container with:

```
# Print TTEs.
/external/BOOM/eval-results-folder.py -z out/
# Printe ITEs.
/external/BOOM/eval-results-folder.py -t out/
```

(E5): Fuzzing Campaign on SmallBoom [< 1 compute-hour]: Run a fuzzing campaign on SmallBoomCore for Meltdown and Spectre-v1 (used to compare against SpecDoctor).
Preparation: Inside of the Docker container, build BOOM in the SmallBoom configuration:

```
phantom-trails build --config SmallFuzzConfig -j<JOBS>
```

Modify `expected_findings` to match SpecDoctor.

```
# Copy old list.
cp expected_findings expected_findings_all
# Create new list.
cat expected_findings_all | head -n 2 > expected_findings
# Only Spectre v1 and Meltdown should be remaining.
```

Finally, remove old results and fuzzing queue.

```
rm -rf out/*
rm .cur_input*
```

Execution: Run the fuzzer

```
phantom-trails fuzz --config SmallFuzzConfig
```

To kill the fuzzer, same instructions as Experiment E4 apply.

Results: TTEs reported in Table 3 of the paper (rightmost column) have been observed on machine **M2** described in Section A.2.3. Statistics of the campaign can be printed as described in Experiment E4.

(E6): Impact of Fuzzing Optimizations [40 minutes + 24 compute-hours]: Remove optimizations from the fuzzer and verify their effect on vulnerability discovery.
Preparation: To run a fuzzing campaign without optimizations you will need to apply the patches located in `phantom-trails/eval-patches`. In principle it is possible to apply one patch at a time and re-run the fuzzing campaign. The most evident effect should be visible by removing both control- and data-flow optimizations ("Basic" column in Table 2).

```
# Exit from the container.
(container) exit
cd <PHANTOM_TRAILS_TOP_FOLDER>
# Remove dataflow opts.
git apply eval-patches/no-dataflow-*
# Remove control flow opts.
git apply eval-patches/no-cfg-*
cd Fuzzer
git apply ../eval-patches/fuzzer/no-cfg-*
```

Finally, re-build the container. This can take up to 40 minutes.

```
cd BOOM
./start.sh
```

Execution: From within the newly built container, run the fuzzer

```
phantom-trails fuzz
```

To kill the fuzzer, same instructions as Experiment E4 apply.
Results: After a 24-hours run it is highly probable that only Spectre-v1 and Meltdown will be found by the tool, as reported in Table 2 of the paper ("Basic" column).

(E7): Fuzzing Campaign with Taint Feedback [40 minutes + 24h to 48 compute-hours]: Run a full fuzzing campaign on MediumBoomCore with the Taint Feedback and verify TTEs.
Preparation: To run a fuzzing campaign with the taint feedback you will need to re-build the instrumented simulation. Exit from the container and run (from the phantom-trails top folder):

```
# Discard all previous modifications.
git checkout .
# Start container with Taint feedback.
cd BOOM
./start.sh "Taint"
```

This can take up to 40 minutes.
Execution: Run the fuzzer

```
phantom-trails fuzz
```

To kill the fuzzer, same instructions as Experiment E4 apply.
Results: TTEs reported in Table 1 of the paper (right part) have been observed on machine **M1** described in Section A.2.3. Statistics of the campaign can be printed as described in Experiment E4.

A.5 Notes on Reusability

Phantom Trails can be used beyond the scope of this paper to introspect the microarchitectural behavior of RISC-V programs and evaluate different feedback metrics.

- Phantom Trails help

  ```
  phantom-trails -h
  ```

  This should show a summary of the three main commands provided by the `phantom-trails` script: `build`, which can be used to build a specific configuration of the BOOM simulation (default one is `FuzzConfig`), `run`, which can be used to run the detector on a single program, and `fuzz`, which can be used to start a fuzzing campaign.

  `phantom-trails` is a simple python script that wraps the invocation of `make`, `run-FuzzConfig` (verilated simulation entrypoint) and `sim-fuzzer` (LibAFL fuzzer) for *build*, *run* and *fuzz* respectively. The content of the script can be inspected at `BOOM/scripts/phantom-trails` (or in the `/scripts/` folder within the container).

- Compile your own sample: once inside the container, you can run the following

  ```
  # Add RISC-V toolchain to env.
  bash
  source /chipyard/env.sh
  fish

  # Add new ASM file to src.
  cd /Samples/src
  echo "nop" > test.S
  cd /Samples && make

  # Run it with Phantom trails.
  cd /sims/chipyard/verilator
  phantom-trails run /Samples/build/bins/test.bin
  ```

- Debugging and Introspection: all configurations can be built with an optional debug flag

  ```
  # Clean previous sims.
  make clean

  # Build in debug mode.
  phantom-trails build -j<PROCS> --debug
  ```

  This enables the possibility of adding additional printing flags at runtime which can be used to inspect the cycle-by-cyle state of the microarchitecture

  ```
  # Run in verbose mode.
  phantom-trails run --verbose <SAMPLE>
  # Add --logfile out.log to output to a file.

  # Print cycle-by-cycle taint and RoB info.
  phantom-trails run --report <SAMPLE>

  # Inspect specific buffers.
  PRINT_REGFILE=1 PRINT_LSQ=1 phantom-trails run <SAMPLE>
  # See boom-wrapper/src/main/resources/csrc/args.h
  # for other print flags.
  ```

- Initialization: you can change or disable the initialization snippet by providing the `-init` flag to `phantom-trails run` (`-init=/dev/null` to disable)

  ```
  phantom-trails --init=/dev/null <SAMPLE>
  ```

  This will avoid prepending the init snippet to the sample under test. To inspect the state of memory right before execution (e.g., if you want to check the exact addresses of the initialization snippet and the sample-under-test in DRAM) you can run:

  ```
  # This generates <SAMPLE>.bin.memdump
  MEMDUMP=1 phantom-trails <SAMPLE>.bin

  # Disassemble the flat binary.
  riscv64-unknown-elf-objdump                                \
              -b binary -m riscv:rv64 -M no-aliases -D \
              <SAMPLE>.bin.memdump
  ```

- Other metrics: Our Dockerfile by default builds LLVM with MSAN (more precisely, our version of MSAN – BFSAN) but in principle different sanitizers can be provided at this stage. Moreover, we build the fuzzing instrumentation using AFL++, and in principle more coverage maps can be added to the codebase we already provide.

- Other buffers: While we use the Physical Register File as a taint sink, it is possible to move the sink to other buffers as well. For this purpose, you can modify `checkTaintSinks()` in `boom-wrapper/src/main/resources/csrc/Simulator.h`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Place Protections at the Right Place: Targeted Hardening for Cryptographic Code against Spectre v1

Yiming Zhu, Wenchao Huang*, Yan Xiong
University of Science and Technology of China

A Artifact Appendix

A.1 Abstract

This appendix provides an overview of each component of the artifact and describes how to use them to reproduce the results presented in the paper. We offer a complete artifact package that includes source code, workloads and documentation.

The artifact consists of five folders: `bench`, `Poc`, `include`, `src`, and `doc`.

The `src` and `include` folders contain the source code for our proposed hardening techniques: LightSLH, LightFence, and LightCut.

The main directory includes a `README.md` file that provides details about key dependencies, instructions for compiling the code, and a simple guide on how to use it.

The `doc` folder contains a `README.md`, including descriptions on how to annotate security properties for the code under analysis and how to mark entry functions for analysis. Additionally, it provides two examples to help users understand how to utilize our tools effectively.

The `Poc` folder includes a proof-of-concept implementation of the proposed side channel described in Section 7.2 of the paper.

Our experimental results can be reproduced by following the step-by-step instructions provided in the `README.md` file located in the `bench` folder. This file outlines the process for downloading the required dependencies and workloads (all of which are open-source).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This tool is a code hardening utility designed to enhance code security through static analysis. It does not involve any destructive operations or disable any security mechanisms. Therefore, evaluators can execute this tool without concerns regarding the security of their machines, data privacy, or other ethical issues.

*Corresponding Authors

A.2.2 How to access

The artifact is available at `https://doi.org/10.5281/zenodo.15569395`. It can be downloaded as a zip file, which contains all the necessary files to run the experiments described in the paper.

A.2.3 Hardware dependencies

None. However, the proof of concept described in Section 7.2 has been tested only on an Intel Xeon® CPU E5-2680.

A.2.4 Software dependencies

The artifact has been tested on Ubuntu 22.04 systems. To install our proposed tool (i.e., LightHardening), the following software dependencies are required:

- GCC and G++ (version 12 or higher), CMake (version 3.25 or higher), Make, and Wget. These dependencies can be installed by running: `apt install gcc-12 g++-12 make wget`. Note that the default Ubuntu apt repository may not include the latest version of CMake. It is recommended to use the official installation script for CMake: `wget https://github.com/Kitware/CMake/releases/download/v3.25.0/cmake-3.25.0-linux-x86_64.sh`, `chmod +x cmake-3.25.0-linux-x86_64.sh`, and `sudo ./cmake-3.25.0-linux-x86_64.sh --prefix=/usr/local`. You can verify the installation by running `cmake -version`.
- LLVM and Clang. Our tool is implemented as an LLVM IR pass and supports multiple LLVM versions. However, to reproduce our experimental results, LLVM version 14.0.4 is required. Due to the specific version requirement for LLVM and the directory structure expected by subsequent execution scripts, please refer to the **Install LLVM** section in `bench/README.md` for detailed instructions on installing LLVM 14.0.4. Note that installing LLVM can be time-intensive, potentially requiring 1-2 hours to complete.

To reproduce our experiments, the following additional software dependencies are required:

- Python 3, Git. These dependencies can be installed by running: `apt install python3 git python3-pip`.
- Dependencies for LLSCT: gperftools, libunwind, ninja, pkg-config, and Python packages: pandas, seaborn. These dependencies can be installed by running: `apt install google-perftools libunwind-dev ninja-build pkg-config` and `pip3 install pandas seaborn`.
- LLSCT. Please refer to the **Install LLSCT** section in `bench/README.md` for detailed instructions on installing LLSCT. Note that LLSCT is a tool directly modified from LLVM, and its installation may require 1-2 hours to complete.
- Google Benchmark. Please refer to the **Install Google Benchmark** section in `bench/README.md` for detailed instructions on installing Google Benchmark.

A.2.5 Benchmarks

We use OpenSSL, Libsodium, NaCL, and PQClean as our workloads. Enter the `bench` folder and run `bash collect_source_code.sh` to prepare these workloads. The script is pre-configured with the specific versions of the libraries used. The downloaded cryptographic source code will be located in the `bench/crypto_code` directory.

A.3 Set-up

A.3.1 Installation

To set up the environment, enter to the bench folder of LightHardening and follow the **Build LightHardening** part in `bench/README.md` for build our tools.

Upon successful execution, the `libSpectrePass.so` file will be generated in the `build` folder. This file represents the LLVM IR pass that implements the proposed hardening techniques.

Next, navigate to the `bench/wrapper` folder and execute `bash compile_wrap.sh`. This step prepares the `.ll` files required for subsequent analysis.

A.3.2 Basic Test

Navigate to the `bench/wrapper` folder and execute the following command: `bash compile.sh aes 16 OpenSSL3.3.0 AES`.

This command compiles the OpenSSL 3.3.0 AES workload with all the hardening tools used in our experiments, including LightSLH, LightFence, LightCut, SSLH, Fence, and LLSCT.

Upon successful execution, the following files will be generated in the `bench/wrapper/OpenSSL3.3.0/aes` folder: `aes_test_LightSLH`, `aes_test_LightFence`, `aes_test_LightCut`, `aes_test_SSLH`, `aes_test_fence`, `aes_test_llsct`, and `aes_test_origin`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The analysis information of LightSLH, as illustrated in Figure 6 of the paper.
(C2): The overheads of various protection tools are compared, as shown in Figure 7 of the paper.
(C3): A one-bit side channel is constructed using the assignment within a loop, as described in Section 7.2 of the paper.

A.4.2 Experiments

(E1): Reproduce C1. [5 compute-minutes]
Preparation: Ensure all dependencies are installed and the code is compiled as described in the **Set-up** section. Navigate to the `bench/wrapper` folder.
Execution: Execute the following command: `bash bench.sh compile`
Results: The analysis information for each workload, corresponding to Figure 6 in our paper, including runtime and processed instruction data, will be saved in `bench/wrapper/compile_information.txt`.

(E2): Reproduce C2. [30 compute-minutes]
Preparation: Ensure E1 has been completed successfully, and navigate to the `bench/wrapper` folder.
Execution: Execute the following commands sequentially:
```
bash bench.sh run
python3 stat_bench.py > performance.txt
```
Results: The runtime of the code generated by different defense methods for each workload will be recorded in `wrapper/result.json`. Additionally, `wrapper/performance.txt` will summarize the overhead introduced by different defense mechanisms compared to unprotected code. Note: The measurement of CPU time may vary depending on processor environment, cache settings, and other factors. As a result, the obtained results may exhibit slight differences compared to those presented in Figure 7.

(E3): Reproduce C3. [1 compute-minutes]
Preparation: Navigate to the `Poc` folder and ensure that LLVM and Clang are installed.
If you have installed LLVM 14.0.4 as described in Section A.2.4, you can add the LLVM binary directory to your `PATH` by running: `export PATH=$(realpath ../bench/llvm-project-14.0.4.src/install/bin):$PATH`. Alternatively, you can use `apt install clang` to install a different version of Clang, as this part of the reproduction does not require a specific Clang version.
Execution: Follow the instructions in the `Poc/README.md` file to run the proof-of-concept code.

Results: When compiled without forbidden branch prediction, there is approximately an 80% chance of obtaining output consistent with the input when running the Python script in the `Poc` folder. In contrast, when compiled with forbidden branch prediction, the probability drops to 50%, which is equivalent to random guessing.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Encarsia: Evaluating CPU Fuzzers via Automatic Bug Injection

Matej Bölcskei
ETH Zurich

Flavien Solt
ETH Zurich

Katharina Ceesay-Seitz
ETH Zurich

Kaveh Razavi
ETH Zurich

A Artifact Appendix

A.1 Abstract

We provide the source code of ENCARSIA along with a Docker-based evaluation setup to facilitate reproducing the results presented in our accompanying paper and to support ENCARSIA's deployment for evaluating future CPU fuzzers. These artifacts demonstrate that ENCARSIA is a fully functional tool capable of injecting bugs into CPUs, formally verifying their architectural observability, and leveraging the generated bugs to assess fuzzers.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

ENCARSIA is a bug injection and fuzzer benchmarking tool that does not pose any security risks, as it is not designed to attack the evaluation system. However, its high degree of parallelism, combined with the significant computational cost of formal verification and RTL simulation, can overwhelm system resources and potentially cause crashes.

A.2.2 How to access

Our artifacts are accessible for permanent access on Zenodo at `https://doi.org/10.5281/zenodo.14664723`. Alternatively, they can be accessed via GitHub at `https://github.com/comsec-group/encarsia`.

A.2.3 Hardware dependencies

ENCARSIA runs on any standard hardware, but requires substantial computing resources to handle the demands of formal verification, RTL simulation, and to take full advantage of parallelization for accelerating the experiments. We therefore recommend a machine with at least 32 CPU cores, 256 GB of main memory, and 512 GB of disk space. For detailed requirements of each experiment, refer to their descriptions in Section A.4.2.

A.2.4 Software dependencies

The following software dependencies are required for running ENCARSIA:

- **`docker`**: to run the experiments in the provided Docker environment.
- **`make`**: for automating tasks like building the Docker image.
- **`tar`**: to extract the EnCorpus dataset, which is provided as a .tar.gz archive.
- **`python3`**: for parsing the results of our bug survey.

If you prefer to run the experiments outside the Docker setup, the Dockerfile can serve as a reference for the necessary software dependencies.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Clone the artifact repository using the GitHub link provided in Section A.2.2, or download it from the permanent access link to our artifacts on Zenodo. After cloning the repository, navigate to the root directory of the repository and run `make pull` to fetch the pre-built Docker image. Alternatively, you can run `make build` to build the Docker image locally.

After obtaining the Docker image, use `make run` to start a container from the image. Note the container ID displayed in the terminal output. You can use it later to restart and attach to the container for further experiments with `docker start <container_id> && docker attach <container_id>`.

A.3.2 Basic Test

Make sure the Docker container is running and attached. Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H ibex rocket boom -p 30` to confirm that ENCARSIA is functioning correctly and parsing the EnCorpus dataset as expected.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Bugs in open-source CPUs often stem from two simple syntactic transformations: mix-ups of signals or logic expressions and errors in conditional statements. This is proven by the bug survey **(E1)** described in Section 4 of the paper whose results are reported in Table 4.

(C2): ENCARSIA can rapidly inject a large number of diverse, realistic bugs into CPUs of varying complexity and design paradigms. This is proven by the injection experiment **(E2)** described in Section 7.1 of the paper whose results are reported in Table 5.

(C3): ENCARSIA can formally prove the architectural observability of bugs within a practical timeframe using computational resources typical for hardware design and verification. This is demonstrated by experiment **(E3)**, which follows a similar approach to the verification experiment described in Section 7.1 of the paper whose results are reported in Table 6. To make our verification framework more accessible, we provide a fully open-source version based on Yosys.

(C4): Instruction-granular bug detection mechanisms do not demonstrate greater potential for detecting bugs. This is demonstrated by the instruction-granular bug detection evaluation **(E4)** described in Section 8.1 of the paper whose results are reported in Table 8.

(C5): The hardware-specific structural coverage metrics, advertised as central by many fuzzers, are of little help in detecting bugs. This is demonstrated by the coverage metrics evaluation **(E5)** described in Section 8.2 of the paper whose results are reported in Table 9.

(C6): The fuzzer seeds are a key factor that determines which bugs will eventually be detected. This is demonstrated by the seed program experiment **(E6)** described in Section 8.3 of the paper whose results are reported in Table 10.

A.4.2 Experiments

All experiments are intended to be run within the provided Docker environment to ensure consistency and reproducibility. Before running each experiment, ensure that your system has sufficient computing resources available to meet the requirements specified for the experiment.

(E1): [Survey (Section 4)] [5 human-minutes]: The results of our bug survey are available in `survey/classification/synthetic.json` and `survey/classification/natural.json` within the main artifacts repository (not Docker image).
Execution: Use `survey/classification/plot.py` to display the results of the survey.
Results: The results of the survey confirm that all identified observable bugs indeed do fall into one of the two categories.

(E2): [Injection (Section 7.1)] [5 human-minutes + 20 compute-minutes + 100GB disk + 8GB memory]: This experiment exactly replicates the injection experiment described in Section 7.1 of the paper.
Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/Injection -H ibex rocket -p 30`. Optionally, add `boom` to the `-H` option to inject bugs into BOOM, but note that this requires up to 512GB of disk space.
Results: This experiment injects around 1000 Signal Mix-ups and 1000 Broken Conditionals per CPU. The resulting `host.v` files and injection logs can be found in the experiment directory at `/encarsia-meta/out/Injection`. A summary of the injection results, similar to Table 5 in the paper, is printed to the terminal. We expect the summary table to closely match the one presented in the paper.

(E3): [Bug Verification] [5 human-minutes + 1 compute-hour + 6GB disk + per-process memory (device dependent: 4 GB for Ibex, 8 GB for Rocket, 32 GB for BOOM)]: This experiment closely replicates the verification experiment outlined in Section 7.1 of the paper, with the main difference being our use of a fully open-source verification setup based on Yosys. Additionally, we limit the experiment to bugs from the EnCorpus bug set to reduce the duration of the experiment.
Execution: Assess your system's computing resources to determine the number of parallel processes it can support for this experiment. Then, navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H ibex rocket boom -p NUM_PROC -Y`, replacing `NUM_PROC` with the number of parallel processes. You can also execute the experiment on one device at a time by running `python encarsia.py -d out/EnCorpus -H DEVICE -p NUM_PROC -Y` for each of the three devices. This ensures optimal memory usage, despite varying memory requirements across devices.
Results: This experiment generates formal proofs of architectural observability for the EnCorpus bugs using the Yosys setup. The resulting verification log (`yosys_verify.log`) and proof of observability (`yosys_proof.S`) can be found in the EnCorpus experiment directory at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the verification results, similar to Table 6 in the paper, is printed directly to the terminal. Note that EnCorpus was verified using the JasperGold setup, which is more robust and powerful. As a result, not all EnCorpus bugs are expected to verify successfully using Yosys. We nevertheless expect Yosys to verify most of the EnCorpus bugs in the simpler CPUs (Ibex and Rocket) and a smaller subset in the more complex BOOM. Furthermore, we expect the average verification time per bug to be similar to the values

reported in Table 6 of the paper.

(E4): [Instruction-granular bug detection evaluation] [5 human-minutes + 10 compute-hours + 100GB disk + 4GB memory per parallel process]: This experiment closely replicates the fuzzing experiment outlined in Section 8.1 of the paper, with the main difference being the reduced fuzzing duration of 30 minutes.

Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H rocket boom -p 30 -F no_cov_difuzzrtl no_cov_processorfuzz`.

Results: This experiment generates two key outputs: a fuzzing log stored in `fuzz.log`, and the bug detection results (after filtering out false positives) in `check_summary.log`. Both files are located within the corresponding fuzzer directories at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the fuzzing results, similar to Table 8 in the paper, is printed directly to the terminal.

We expect the results to match those presented in the paper, except for Rocket Signal Mix-up 1 and BOOM Signal Mix-ups 9 and 14. These bugs are detected by DifuzzRTL and Processorfuzz in the Docker setup, but remain undetected in the bare-metal setup used to generate the data presented in the paper despite several fuzzing re-runs. We suspect this discrepancy stems from differing versions of dependencies, such as Spike or Verilator, used internally by the fuzzers. However, the lack of transparency regarding which tools are used and their specific versions makes it difficult to determine the exact cause. Despite this discrepancy, we firmly believe that the major claims of the paper remain valid.

(E5): [Coverage metrics evaluation] [5 human-minutes + 10 compute-hours + 100GB disk + 4GB memory per parallel process]: This experiment closely replicates the fuzzing experiment outlined in Section 8.2 of the paper, with the main difference being the reduced fuzzing duration of 30 minutes.

Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H rocket boom -p 30 -F difuzzrtl processorfuzz`.

Results: This experiment generates two key outputs: a fuzzing log stored in `fuzz.log`, and the bug detection results (after filtering out false positives) in `check_summary.log`. Both files are located within the corresponding fuzzer directories at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the fuzzing results, similar to Table 9 in the paper, is printed directly to the terminal.

As in the previous experiment, we observe discrepancies between the Docker and bare-metal setups for Rocket Signal Mix-up 1 and BOOM Signal Mix-ups 9 and 14.

(E6): [Seed program evaluation] [5 human-minutes + 90 compute-minutes + 10GB disk + 4GB memory per parallel process]: This experiment closely replicates the fuzzing experiment outlined in Section 8.3 of the paper, with the main difference being the reduced fuzzing duration of 30 minutes.

Note that this experiment reuses the results of the DifuzzRTL evaluation from experiment **(E5)**, if available, which reduces computation time and disk space requirements. If the results of experiment **(E5)** are not available, the requirements are approximately half those of experiment **(E5)**.

Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H rocket boom -p 30 -F difuzzrtl cascade`.

Results: This experiment generates two key outputs: a fuzzing log stored in `fuzz.log`, and the bug detection results (after filtering out false positives) in `check_summary.log`. Both files are located within the corresponding fuzzer directories at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the fuzzing results, similar to Table 10 in the paper, is printed directly to the terminal.

As in the previous experiments, we observe discrepancies between the Docker and bare-metal setups for Rocket Signal Mix-up 1 and BOOM Signal Mix-ups 9 and 14 on DifuzzRTL. However, the results for Cascade exactly match those reported in the paper.

A.5 Troubleshooting Guide

- **OOM Errors:** Ensure your system meets the minimum memory requirements specified for the experiment by reducing the number of parallel processes (`-p NUM_PROC`).
- **Running Out of Disk Space:** Verify that your system has sufficient disk space before starting an experiment. Clean up old experiment outputs if necessary.
- **Badly Terminated Parallel Experiments:** If an experiment is terminated ungracefully, some processes may remain running in the background. Use `ps` to identify and terminate any processes stuck in an infinite loop.

A.6 Notes on Reusability

ENCARSIA can be easily extended to support additional CPUs by creating a new `EncarsiaConfig` instance in `/encarsia-meta/config.py`. For more information, see the README in the ENCARSIA repository.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions

Sandro Rüegge
ETH Zurich

Johannes Wikner
ETH Zurich

Kaveh Razavi
ETH Zurich

A Artifact Appendix

A.1 Abstract

This artifact appendix provides instructions to reproduce the results in *Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions*. Our paper introduces Branch Predictor Race Condition (BPRC), an event-misordering effect in branch predictors. We uncover multiple instances of this effect on recent Intel processors and investigate the cause. To demonstrate the impact of these vulnerabilities, we built an end-to-end attack on Ubuntu 24.04, running a recent Linux kernel with default mitigations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Some of the experiments require loading special kernel modules. *kmod_ap* allows any user process to execute arbitrary code with supervisor privileges. *kmod_spec_ctrl* provides persistent control over the speculation control register. These modules compromise the system's security but can be manually removed using `sudo rmmod ap` and `sudo rmmod spec_ctrl`.

Furthermore, the provided ansible scripts modify the system configuration (i.e. GRUB, initramfs) without any additional interaction. While the scripts make an effort to return the system to it's previous state, **please exercise extreme caution when using the ansible scripts** to ensure they do not operate outside of your expectations. Additionally, the ansible scripts will reboot the target system when necessary which makes localhost unsuited to run these scripts against.

A.2.2 How to access

The artifacts are available on zenodo at https://doi.org/10.5281/zenodo.14636810 and on GitHub at https://github.com/comsec-group/bprc (once the embargo ends on May, 13th). We provide the artifact evaluators with a ssh key in the hotcrp submission to access the git repository privately during the embargo.

A.2.3 Hardware dependencies

In our paper, we evaluated a wide range of recent microarchitectures. The exact microarchitectures used in our paper are:

- Core i7-14700K (Raptor Cove, Gracemont)
- Core i7-13700K (Raptor Cove, Gracemont)
- Core i7-12700K (Golden Cove, Gracemont)
- Core i7-11700K (Cypress Cove)
- Core i7-10700K (Skylake)
- Core i9-9900K (Skylake)
- Ryzen 9 9900X (Zen 5)
- Ryzen 7 7700X (Zen 4)
- Google Tensor (Cortex-X1)
- Google Tensor (Cortex-A76)

The microarchitectural experiments focus mostly on the Intel processors with the exception of the initial asynchronous branch predictor experiment. The attack requires Ubuntu 24.04 with kernel package version 6.8.0-47-generic. While we developed the attack specifically for a Core i7-13700K, other recent Intel processors should work as well, albeit potentially less reliably.

A.2.4 Software dependencies

We use Ubuntu 24.04, 22.04, or 20.04 to compile and run the experiments on our test machines. Our compilation scripts expect Ubuntu packages `build-essential`, `msr-tools`, `clang`, `git`, `libtirpc-dev`, and the kernel headers (`linux-headers-generic`) to be present. The python analysis scripts require `python3` with `matplotlib`.

A.2.5 Benchmarks

Our mitigation performance evaluation employs UnixBench [1] and lmbench [2].

[1] https://github.com/kdlucas/byte-unixbench
[2] https://github.com/intel/lmbench

A.3 Set-up

A.3.1 Installation

Download the artifacts using one of the links in Section A.2.2. Install the dependencies as per the *README.md* file. On Ubuntu 24.04 you can use the following commands:

```
sudo apt install python3 python3-pip \
   build-essential git clang \
   linux-headers-$(uname -r) msr-tools \
   libtirpc-dev
pip install matplotlib
```

A.3.2 Basic Test

There is a special experiment to help you verify your setup in `experiments/exp-test-setup`. Please follow the `README.md` in the same directory to run the setup test.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Branch predictor updates on recent Intel processors can occur after a measurable delay. This is proven by the experiment (E1) whose results are reported in Section 5.1 and Figure 3.

(C2): As a result, branch predictions can be injected from user to kernel ((E2), Table 2), from guest to host ((E3), Section 5.3) and across IBPB ((E4), Section 5.3).

(C3): We can use these vulnerabilities to build an end-to-end attack to leak arbitrary kernel memory and '/etc/shadow' on an Intel Raptor Lake system with Ubuntu 24.04 ((E10), Section 7).

A.4.2 Experiments

(E1): [exp-btb-delay] [5 human-minutes + 2 compute-hours]: Experiment to determine IP-based branch predictor (BTB) insertion delay (C1).
Execution: Please follow the instructions located at `experiments/exp-btb-delay/README.md`
Results: On Intel processors, we observe a high number of mispredictions when the delay (number of NOPS) is low and vice versa

(E2): [exp-leak-supervisor] [5 human-minutes + 5 compute-minutes]: Experiment to determine if $BPRC_{U\rightarrow K}$ is possible on recent Intel processors (C2).
Execution: Please follow the instructions located at `experiments/exp-leak-supervisor/README.md`
Results: We observe a success percentage above the noise level on at least two of the three tested instruction on recent Intel processors

(E3): [exp-leak-hypervisor] [5 human-minutes + 5 compute-minutes]: Experiment to determine if $BPRC_{G\rightarrow H}$ is possible on recent Intel processors (C2).
Execution: Please follow the instructions located at `experiments/exp-leak-hypervisor/README.md`
Results: We observe a success percentage above the noise level on at least two of the three tested instruction on recent Intel processors

(E4): [exp-leak-ibpb] [5 human-minutes + 5 compute-minutes]: Experiment to determine if $BPRC_{IBPB}$ is possible on recent Intel processors (C2).
Execution: Please follow the instructions located at `experiments/exp-leak-ibpb/README.md`
Results: We observe a success percentage above the noise level on at least two of the three tested instruction on recent Intel processors

(E5): [exp-ibp-insertion] [5 human-minutes + 5 compute-minutes]: Experiment to determine if a prediction is inserted into the BTB and IBP the first time it is seen.
Execution: Please follow the instructions located at `experiments/exp-ibp-insertion/README.md`
Results: We ran this experiment on the Intel performance (P-Cores), Cypress Cove and Skylake cores. For 'Random History' we expect many hits on the BTB (>90%) and very few on the IBP (<1%). For 'Matching History' we expect few hits on the BTB (<10%) and many hits on the IBP (>90%).

(E6): [exp-leak-supervisor-discern] [5 human-minutes + 5 compute-minutes]: Experiment to determine if $BPRC_{U\rightarrow K}$ is caused by the BTB or IBP on Intel.
Execution: Please use instructions at `experiments/exp-leak-supervisor-discern/README.md`
Results: We expect to see hits on the BTB above the noise level while there are no hits above noise level on the IBP.

(E7): [exp-syscall-split] [5 human-minutes + 3 compute-hours]: Experiment to determine the effect of timing on BPI.
Execution: Please follow the instructions located at `experiments/exp-syscall-split/README.md`
Results: We expect to see the number of hits in the kernel decrease with increasing number of NOPs while the number of hits in user increases with increasing number of NOPs.

(E8): [exp-leak-rounds] [5 human-minutes + 5 compute-minutes]: Test reliability of BPI when repeatedly attacking the same victim branch without modifying history with and without `BHI_DIS_S`.
Execution: Please follow the instructions located at `experiments/exp-leak-rounds/README.md`
Results: With only eIBRS we expect a decreasing success rate over successive repetitions. With `BHI_DIS_S` enabled, we expect the success rate to increase over the first two repetitions and then stay high (>90%).

(E9): [exp-bhi-dis-s] [5 human-minutes + 5 compute-minutes]: Experiment to determine how `BHI_DIS_S` is implemented on recent Intel processors.

Execution: Please follow the instructions located at `experiments/exp-bhi-dis-s/README.md`
Results: We expect a low number of mispredictions when there is no mitigation active (<10%) and a high number of mispredictions when `BHI_DIS_S` is active.

(E10): [exp-end2end] [5 human-minutes + 10 compute-minutes]: End-to-end attack against Ubuntu 24.04 kernel version 6.8.0-47-generic on an Intel Raptor Lake processor (C3).
Execution: Please follow the instructions located at `experiments/exp-end2end/README.md`
Results: We expect the attack to work successfully in a high number of cases (>90%) with a leak rate of over 5KiB/s in the median.

(E11): [exp-benchmark-mitigations] [5 human-minutes + 24 compute-hours]: Measure mitigation performance overhead.
Execution: Please use instructions at `experiments/exp-benchmark-mitigations/README.md`
Results: We observed a slowdown for all mitigations. The microcode performs best, `IPRED_DIS_S` second and retpoline third.

A.5 Notes on Reusability

Our experiments were developed to work well on the evaluated x86 processors but the methodologies can be applied to other processors and microarchitectures as well. Furthermore, the kernel modules and library code provided in `experiments/uarch-research-fw` are useful to hardware reverse engineering in general.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix of "GraphAce: Secure Two-Party Graph Analysis Achieving Communication Efficiency"

Jiping Yu [1,2,*], Kun Chen [2], Yunyi Chen [1,2,*], Xiaoyu Fan [1,2,*], Xiaowei Zhu [2], Cheng Hong [2], and Wenguang Chen [1,2]

[1] Tsinghua University
[2] Ant Group

A Artifact Appendix

A.1 Abstract

The paper proposes GraphAce, an efficient secure two-party graph analysis framework. For each iteration, it achieves low complexities of $\Theta(|V|)$ communication, breaking the $\Omega(|V|+|E|)$ lower bound of previous secure solutions, and $\Theta(|V|+|E|)$ computation, which is the same as insecure methods. Evaluations show that GraphAce exceeds previous methods by up to tens of thousands of times in speed and saves up to 99.99% communication, depending on the application and the network. This artifact is anticipated to satisfy: (Available) the source code of GraphAce and other materials mentioned in the "Open Science" Section are made available; (Functional) the artifact contains the necessary instructions and scripts for the software to be executed successfully; (Reproduced) the artifact can reproduce the main claims of the paper, i.e. GraphAce outperforms the existing work GraphSC in terms of communication and execution time, for various graphs, applications, and network configurations.

One-stop quick start (for Ubuntu systems):

```
sudo apt install docker.io xz-utils
wget -O graphace.tar.xz "https://zenodo.org/\
records/15009743/files/graphace.tar.xz"
cat graphace.tar.xz | xz -d | tar x
cd graphace
sudo docker build -t graphace .
sudo docker run \
  --cap-add SYS_NICE --cap-add NET_ADMIN \
  -v ./results:/app/results -it graphace \
  python3 run.py --preset toy
```

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

*Work is done during the research internship at Ant Group.

A.2.2 How to access

The artifact is available at https://zenodo.org/records/15009743. (Note that this is an updated version of https://zenodo.org/records/14720419.)

A.2.3 Hardware dependencies

The artifact requires a machine that meets these requirements:

- **CPU**: x86-64. For more precise replication of the authors' results, high-performance server CPUs are recommended (the authors employed two Intel Xeon 8380 CPUs, providing a total of 80 cores or 160 threads). However, PC or laptop CPUs are generally sufficient to confirm the artifact's operational functionality.
- **RAM**: varies with the experiment. The most extensive experiment requires 768 GB of memory. However, 128 GB is sufficient to validate the major claims. The minimal RAM necessary to conduct a meaningful comparative experiment is 32 GB.
- **Disk**: at least 32 GB free space.
- **OS**: Linux (distribution is not restricted).
- **Network**: an Internet connection.

A.2.4 Software dependencies

The artifact is available in `.tar.xz` format, necessitating decompression capabilities of the evaluator. Docker is required to evaluate this artifact. For example, on Ubuntu systems, the required dependencies can be installed using the command `sudo apt install docker.io xz-utils`.

Docker handles all specific dependencies related to compilation, thus freeing the evaluator from these concerns.

A.2.5 Benchmarks

The artifact operates independently of external data sets. It relies solely on synthetic data that the program generates according to the specified data size.

A.3 Set-up

A.3.1 Installation

For artifact installation, run the commands listed below, assuming that `graphace.tar.xz` has been downloaded to the present working directory.

```
cat graphace.tar.xz | xz -d | tar x
cd graphace
sudo docker build -t graphace .
```

This is expected to finish in 20 minutes. Should the installation encounter issues owing to an external factor (such as an unstable Internet connection preventing the download of a dependency), it may be beneficial to clear the Docker build cache and attempt the process again.

A.3.2 Basic Test

To perform a basic test, execute the following:

```
sudo docker run \
  --cap-add SYS_NICE --cap-add NET_ADMIN \
  -v ./results:/app/results -it graphace \
  python3 run.py \
    --framework graphace --graph scale14 \
    --application pr --network 1DC
```

This command executes the PageRank application using GraphAce on the scale-14 graph over a 1-DC network. The completion is expected in one minute. Upon success, it provides information on execution time and the amount of communication, indicating the artifact's successful installation.

Should you be interested, the command is thoroughly explained as follows.

- The `--cap-add` parameters are essential for the artifact's proper functionality. Specifically, `SYS_NICE` permits the regulation of NUMA accesses within the Docker container, and `NET_ADMIN` facilitates the manipulation of bandwidth and network latency. These networking effects are confined to the container and do not impact programs external to the Docker environment.
- The `-v` parameter ensures that running logs are accessible in the `results` directory outside the container.
- The last four parameters are configurable, which is especially useful if the evaluator is interested in a specific experiment but does not wish to conduct a whole experiment preset (E1 to E5) below. In detail,
  * `--framework` specifies the framework to run. It can be `graphace` or `graphsc`.
  * `--graph` specifies the input graph. GraphSC supports `scale14`, `scale17`, and `scale20`, while GraphAce supports `scale{14,17,20,24,27}` and `scale{14,17,20}{II,III,IV,V}` (for Table 3 and Figure 8). There is no `scale20I`, for example, because it is actually the same as `scale20`.
  * `--application` specifies the application to run. It can be `pr`, `spmv`, `sslp`, `wcc`, or `mssp`.
  * `--network` specifies the network configuration. It can be `1DC`, `interDC`, or `interContinent`.

A.4 Evaluation workflow

A.4.1 Major Claims

The artifact is expected to support the major claims presented in the paper, as outlined below.

(C1): GraphAce decreases the amount of communication per iteration compared to GraphSC, in multiple applications and varying graph scales. This is demonstrated by any of the experiments (E1) to (E5), whose results are illustrated in Figure 6 of Section 6.2.

(C2): GraphAce decreases iteration time in most scenarios compared to GraphSC, in multiple applications, varying graph scales, and different network conditions. This is demonstrated by any of the experiments (E1) to (E5), whose results are illustrated in Figure 7 of Section 6.2.

(C3): GraphAce consistently provides communication and time savings compared to GraphSC for graphs with the same scale but varying average degrees, as observed in the two representative applications, PR and MSSP. This is demonstrated by any of the experiments (E1) to (E5), whose results are illustrated in Figure 8 of Section 6.3.

A.4.2 Experiments

Rather than presenting a separate experiment for each figure, we present five "presets," each of which can at least partially support all three major claims. The rationale behind this is the substantial time required to fully reproduce each subfigure; the quickest takes 6 hours, whereas the slowest takes up to 2 days, and we have over twenty subfigures. Therefore, we offer five experimental presets with completion times ranging from 10 minutes to 10 days. This allows the evaluator to select an appropriate preset based on their compute-time budget, to derive the most meaningful results within the constraints of limited execution time.

Experiment preset	Compute time[1]	Min. RAM	Support (C1)?	Support (C2)&(C3)?
(E1) Toy	10 min.	32G	Limited	Limited
(E2) Small	2.5 hours	32G	Partial	Partial
(E3) Medium	15 hours	128G	Yes	Partial
(E4) Large	75 hours	128G	Yes	Yes
(E5) XLarge	10 days	768G	Yes, better speedup ratio	

Table 1: Experiment presets

[1]The compute time is estimated on the basis of the hardware used by the authors, specifically two Intel Xeon 8380 CPUs with a total of 80 cores or 160 threads. If the evaluator uses a lower performance machine, it is expected that additional computation time will be required for the experiments.

Table 1 provides data on the five presets available. Larger presets necessitate increased computational time and RAM but can yield more thorough results (e.g., encompassing more applications and network configurations) and an improved ratio, as GraphAce displays enhanced complexity and benefits more from larger inputs. Therefore, selecting the largest feasible preset is advised to more accurately replicate the results presented in the paper. Should you need to stop execution prematurely, rest assured that an intermediate CSV file containing all results up to that moment is generated at the end of each run, enabling the evaluator to review partial findings even if the script does not run to completion.

In the presets "Toy" and "Small", the resulting data for Figure 8 refer to graphs of scale 14, rather than scale 20 in the paper. Analogously, the "Medium" and "Large" presets correspond to a scale of 17. *The speedup ratio might be less advantageous than reported in the paper if the utilized CPU exhibits lower performance than the one used by the author, or if only smaller presets are executed.*

(E1): [Toy preset] [**10 compute-minutes + 32 GB RAM**]: This preset aims to provide a quick glimpse of the results. It is expected to verify (C1) to (C3), but only on a single application (PR), a single network setting (1-DC), and limited graph sizes (up to scale 20 for GraphAce and scale 14 for GraphSC).
How to: Execute the command and wait for the results.
Preparation: No specific preparation is needed other than to complete the basic test.
Execution: Execute the following command:

```
sudo docker run \
  --cap-add SYS_NICE --cap-add NET_ADMIN \
  -v ./results:/app/results -it graphace \
  python3 run.py --preset toy
```

Results: The results will appear in the `results` directory as a CSV file, which can indicate that (part of C1) GraphAce decreases the communication amount for PR, (part of C2) GraphAce shortens the execution time for PR 1-DC, and (part of C3) this benefit remains consistent across graphs with varying average degrees.

(E2): [Small preset] [**2.5 compute-hours + 32 GB RAM**]: This preset aims to provide a slightly more comprehensive view of the results. It is expected to verify (C1) to (C3), but only on two representative applications (PR and MSSP), two network settings (1-DC and Inter-DC), and limited graph sizes (up to scale 20 for GraphAce PR and scale 14 for the others).
Execution: `sudo docker run \`
`  --cap-add SYS_NICE --cap-add NET_ADMIN \`
`  -v ./results:/app/results -it graphace \`
`  python3 run.py --preset small`
Results: The results will appear in the `results` directory as a CSV file, which can indicate that (part of C1) GraphAce decreases the communication amount for PR and MSSP, (part of C2) GraphAce shortens the execution time for PR/MSSP 1-DC/Inter-DC, and (part of C3) this benefit remains consistent across graphs with varying average degrees.
It should be noted that GraphAce could potentially operate at a slower speed compared to GraphSC for MSSP 1-DC, particularly for the constrained graph scale 14, as mentioned in the paper.

(E3): [Medium preset] [**15 compute-hours + 128 GB RAM**]: This preset is expected to verify (C1) to (C3) for all the five applications, but only on two network settings (1-DC and Inter-DC), and limited graph sizes (Scale 24 for GraphAce PHE, Scale 17 for GraphAce FHE and GraphSC 1-DC, and Scale 14 for GraphSC Inter-DC).
Execution: `sudo docker run \`
`  --cap-add SYS_NICE --cap-add NET_ADMIN \`
`  -v ./results:/app/results -it graphace \`
`  python3 run.py --preset medium`
Results: The results will appear in the `results` directory as a CSV file, which can indicate that (C1) GraphAce decreases the communication amount, (part of C2) GraphAce shortens the execution time for 1-DC/Inter-DC, and (part of C3) this benefit remains consistent across graphs with varying average degrees.

(E4): [Large preset] [**75 compute-hours + 128 GB RAM**]: This preset is expected to fully verify all major claims (C1) to (C3).
Execution: `sudo docker run \`
`  --cap-add SYS_NICE --cap-add NET_ADMIN \`
`  -v ./results:/app/results -it graphace \`
`  python3 run.py --preset large`
Results: The results will appear in the `results` directory as a CSV file, which fully supports (C1) to (C3).

(E5): [XLarge preset] [**10 compute-days + 768 GB RAM**]: This preset is expected to fully verify all major claims (C1) to (C3). In comparison to (E4), it is anticipated that this will deliver a greater communication saving ratio or an increase in execution time speedup, due to the larger graph scale and GraphAce's complexity improvement.
Execution: `sudo docker run \`
`  --cap-add SYS_NICE --cap-add NET_ADMIN \`
`  -v ./results:/app/results -it graphace \`
`  python3 run.py --preset xlarge`
Results: The results will appear in the `results` directory as a CSV file, which fully supports (C1) to (C3).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: HawkEye: Statically and Accurately Profiling the Communication Cost of Models in Multi-party Learning

Wenqiang Ruan
Fudan University

Xin Lin
Fudan University

Ruisheng Zhou
Fudan University

Guopeng Lin
Fudan University

Shui Yu
University of Technology Sydney

Weili Han*
Fudan University

A Artifact Appendix

A.1 Abstract

We present HawkEye, a static model communication cost profiling framework for multi-party learning (MPL), which is an important technology for utilizing data from multiple parties with privacy preservation. Our artifact includes:

1. the accuracy evaluation of HawkEye.
2. three practical applications of HawkEye.

This artifact reproduces the tables and figures in our paper.

A.2 Description & Requirements

HawkEye enables model designers to get the accurate communication cost of PyTorch-based models in multi-party learning (MPL) frameworks without dynamically running the secure model training or inference processes on a specific MPL framework. For example, model designers can apply HawkEye to statically analyze the communication cost of different components of a CNN model constructed with PyTorch on an MPL framework (e.g., ABY3) without running the secure model training or inference process on the MPL framework. Therefore, with HawkEye, model designers can find the performance bottleneck of models efficiently and then design efficient models for different MPL frameworks.

HawkEye currently supports static model communication cost profiling on ten MPL frameworks, including CrypTFlow2, CrypTen, ABY, SPDZ-2k, ABY3, and Falcon, Delphi, Cheetah, Deep-MPC, SecretFLow-SEMI2K. Model designers can apply HawkEye to statically analyze the communication costs of PyTorch-based models on these ten MPL frameworks.

HawkEye is developed based on the compiler of MP-SPDZ that is implemented by Python. The codebase also includes the virtual machine of MP-SPDZ to reproduce experimental results in the paper.

*Corresponding author: wlhan@fudan.edu.cn

A.2.1 Security, privacy, and ethical concerns

The execution of our artifact would not cause security, privacy, or ethical risks to the machines of evaluators.

A.2.2 How to access

HawkEye can be accessed through two ways: (1) Zenodo. The URL is https://zenodo.org/records/14855032. (2) Github. The URL is https://github.com/wqruan/HawkEye. The above two repositories both include instructions for deploying HawkEye and reproducing our results.

A.2.3 Hardware dependencies

HawkEye does not require any specialized hardware. Our experiments were run on a Linux server equipped with two 32-core 2.30 GHz Intel Xeon CPUs and 512GB of RAM. The OS version is Ubuntu 20.04.

A.2.4 Software dependencies

The evaluation of our artifact needs to be performed on a Ubuntu Server that supports Python 3.8 and C++ 17. For the installation of other third-party software, we describe them in detail in the README documentation of the artifact.

A.2.5 Benchmarks

None.

A.3 Set-up

In this section, we provide information about setting up and running the artifacts.

A.3.1 Installation

We provide instructions on how to install the dependencies and necessary configuration steps in the README documen-

tation of https://github.com/wqruan/HawkEye/tree/main

A.3.2 Basic Test

After installation, you can run the command 'python compile.py -R 60 -Q CryptFlow2 -C -K LTZ,TruncPr autograd_shufflenetv2 –profiling' to check whether the installation is successful. The expected output is:

```
Conv2d
online comm size:38.87GB proportion:86.14%
online comm round:1745  proportion:67.37%
offline comm size:0.0GB proportion:0.00%
offline comm round:0.0  proportion:0.00%
Other linear
online comm size:4.15GB proportion:9.2%
online comm round:599   proportion:23.13%
offline comm size:0.0GB proportion:0.00%
offline comm round:0.0  proportion:0.00%
Linear
online comm size:43.02GB proportion:95.34%
online comm round:2344  proportion:90.5%
offline comm size:0.0GB proportion:0.00%
offline comm round:0.0  proportion:0.00%
Non_linear
online comm size:2.1GB  proportion:4.66%
online comm round:246   proportion:9.5%
offline comm size:0.0GB proportion:0.00%
offline comm round:0.0  proportion:0.00%
profiling time: 0.09888839721679688
Program requires at most:
387620574936 online communication bits
           0 offline communication bits
        2590 online round
           0 offline round
compiling time: 15.798906803131104
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): HawkEye can accurately profile the model communication cost on different MPL frameworks. This is proven by experiments (E1). The result is described in Section 5.2, Appendix B, and Appendix C of the full version of the paper and is illustrated in Tables 1, 2, 6, 7, 8, 9, and Figures 6 and 7.

(C2): HawkEye can be used to find communication bottlenecks of models on MPL frameworks with different security models, choose a proper optimizer for secure model training, and optimize the model computational graph. These are proven by experiments (E2, E3, E4). The result is described in Section 5.5 of the full version of the paper and is illustrated in Tables 4, 5, and Figure 8.

A.4.2 Experiments

(E1) Accuracy of HawkEye (80 human-minutes + 4 compute-hour + 64GB memory): This experiment evaluates the accuracy of HawkEye. We compare the communication cost profiling results outputted by HawkEye and five MPL frameworks (i.e. CrypTFlow2, CrypTen, Delphi, Cheetah, SecretFLow-SEMI2K) to show the accuracy of HawkEye.
Preparation: Build the environment as described in the following link https://github.com/wqruan/HawkEye#build-the-environment.
Execution: Use the scripts described in the following link https://github.com/wqruan/HawkEye#accuracy-of-hawkeye to run the experiments and retrieve the results. Note that the source codes of baselines are stored in other repositories. If you want to run our baselines, you can follow the instructions of REAMDE-HawkEye.md documentation in their repositories (https://github.com/wqruan/MPCFormer-HawkEye and https://github.com/yNotAVAILABLEa/Delphi-HawkEye/tree/main). The README documentation of https://github.com/wqruan/HawkEye/tree/main provides more details.
Results: The experimental results would be stored in text files. The README documentation of https://github.com/wqruan/HawkEye/tree/main provides more details. This experiment supports claim (C1).

(E2) Impact of security models (10 human-minutes + 1 compute-hour + 32GB memory): This experiment shows that HawkEye can help model designers find communication bottlenecks of models on MPL frameworks with different security models. We apply HawkEye to profile model communication cost on four MPL frameworks whose security models are different.
Preparation: Build the environment as described in the following link https://github.com/wqruan/HawkEye#build-the-environment.
Execution: Use the scripts described in the following link https://github.com/wqruan/HawkEye#impact-of-security-models to run the experiments and retrieve the results.
Results: The experimental results will be shown in a figure corresponding to Figure 8 of the full version of the paper. This experiment supports claim (C2).

(E3) Choice of optimizers (10 human-minutes + 1 compute-hour + 32GB memory): This experiment shows that HawkEye can help model designers choose a proper optimizer for secure model training. We apply HawkEye to profile the communication cost of secure model training

processes with two different optimizers.

Preparation: Build the environment as described in the following link https://github.com/wqruan/HawkEye#build-the-environment.

Execution: Use the scripts described in the following link https://github.com/wqruan/HawkEye#choice-of-optimizers to run the experiments and retrieve the results.

Results: The experimental results would be stored in text files. The README documentation of https://github.com/wqruan/HawkEye/tree/main provides more details. This experiment supports claim (C2).

(E4) Computational graph optimization (30 human-minutes + 24 compute-hour + 32GB memory): This experiment shows that HawkEye can improve the effectiveness of classical computational graph optimization in secure model inference. We combine HawkEye with TASO, a classical computational graph optimization method for deep learning models, to effectively reduce the communication overhead of secure model inference.

Preparation: Build the environment as described in the following link https://github.com/wqruan/HawkEye?tab=readme-ov-file#prepare-environment.

Execution: Use the scripts described in the following link https://github.com/wqruan/HawkEye?tab=readme-ov-file#run-experiments to run the experiments and retrieve the results.

Results: We provide a Python script to parse the experimental results. The README documentation of https://github.com/wqruan/HawkEye/tree/main provides more details. This experiment supports claim (C2).

A.5 Notes on Reusability

HawkEye can be easily reused to profile the communication costs of models on different MPL frameworks. Firstly, HawkEye can be easily deployed and run. HawkEye is purely implemented by Python and only depends on three Python libraries (i.e., numpy, matmplotlib, and setuptools) that can be installed by popular package management tools (e.g., pip). Meanwhile, HawkEye can be run by one Python command and the scripts in the folder 'HawkEye/Scripts' include many examples. Secondly, model designers can easily construct models they want to profile based on HawkEye. HawkEye provides an autograd library whose interfaces are fully consistent with PyTorch. Meanwhile, the folder 'HawkEye/Programs/Source' includes many model construction examples. Finally, HawkEye has supported ten popular MPL frameworks. In summary, model designers can easily reuse HawkEye to profile the communication costs of models on different MPL frameworks.

In addition, HawkEye can be extended to new MPL frameworks by adding the communication cost configuration of the MPL frameworks in the 'HawkEye/Compiler/cost_config.py.' Section 3.1 of the full version of the paper describes the communication cost configuration process detailedly. Meanwhile, 'HawkEye/Compiler/cost_config.py' includes many examples of MPL framework communication cost configurations.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: General-Purpose f-DP Estimation and Auditing in a Black-Box Setting

Önder Askin[1], Holger Dette[1], Martin Dunsche[1,*], Tim Kutta[2], Yun Lu[3], Yu Wei[4], Vassilis Zikas[4†]

[1]Ruhr-University Bochum
[2]Aarhus University
[3]University of Victoria
[4]Georgia Institute of Technology

A Artifact Appendix

A.1 Abstract

In this paper we propose new methods to statistically assess f-Differential Privacy (f-DP), a recent refinement of differential privacy (DP) that remedies certain weaknesses of standard DP (including tightness under algorithmic composition). A challenge when deploying differentially private mechanisms is that DP is hard to validate, especially in the black-box setting. This has led to numerous empirical methods for auditing standard DP, while f-DP remains less explored. We introduce new black-box methods for f-DP that, unlike existing approaches for this privacy notion, do not require prior knowledge of the investigated algorithm. Our procedure yields a complete estimate of the f-DP trade-off curve, with theoretical guarantees of convergence. Additionally, we propose an efficient auditing method that empirically detects f-DP violations with statistical certainty, merging techniques from non-parametric estimation and optimal classification theory. The artifact will demonstrate the effectiveness for a range of DP mechanisms. The plug-and-play design allows easy replacement of mechanisms and extension to new ones.

A.2 Description & Requirements

This artifact implements a black-box f-DP estimator using two approaches: a perturbed likelihood ratio (PLRT) test and a classifier-based Baybox estimator. The codebase is tested on an Ubuntu 22.04.5 LTS machine and requires Python 3 with venv, pip, and JupyterLab for execution, alongside R version 4.1.2 and the fdrtool package for subsampling experiments. Pretrained models are provided to avoid expensive retraining, but scripts are available to regenerate them if needed. Both the estimator and auditor can be evaluated using Jupyter notebooks located in the `results/` directory, which reproduce all key experiments and figures from the paper.

*Corresponding author: martin.dunsche@rub.de
†Authors are listed in alphabetical order.

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy or ethical concerns.

A.2.2 How to access

For long-term accessibility, an archived version is hosted on Zenodo at: https://doi.org/10.5281/zenodo.15599462. The link provides full access to the source code, pretrained models, result folders, and example notebooks needed to reproduce all experimental results. Please refer to the Zenodo record for a permanent, citable reference to the evaluated version of the artifact. Furthermore we update a GitHub-repository accessible at: https://github.com/stoneboat/fdp-estimation.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

The artifact requires a standard software stack including Python 3, R (version 4.1.2), and commonly used open-source packages:

- *Python libraries are listed in requirements.txt .*
- *R requires fdrtool .*

A.2.5 Benchmarks

The artifact includes precomputed outputs and models used in the experiments reported in the paper:

- ***Pretrained models** used for the CIFAR-10 experiments are provided in the `trained_models/` directory.*

While training from scratch is computationally expensive, reproducibility is supported via scripts in `sgd_experiment_cnn/`.

- ***Benchmark results*** *for Figures 1–12 are stored in the* `results/` *directory, alongside Jupyter notebooks for reproducing f-DP curve estimation and auditing.*

A.3 Set-up

A.3.1 Installation

The artifact is tested on Ubuntu 22.04.5 LTS, and installation has been verified on a Google Cloud VM instance. The following steps set up both the Python and R environments:

1. **System update (optional but recommended):**

```
sudo apt update sudo apt upgrade -y
```

2. **Install Python environment:**

```
sudo apt install python3-venv python3-pip
python3 -m venv fdp-env
source fdp-env/bin/activate
pip install -upgrade -r requirements.txt
```

3. **Install R for Subsampling experiments:**

```
sudo apt install r-base -y sudo R
```

Inside the R console:

```
install.packages("fdrtool")
```

4. **Configure Jupyter for interactive use:**

```
python -m ipykernel install -user
-name=fdp-env -display-name "Python
(fdp-env)"
jupyter lab -ip=0.0.0.0 -port=8888
-no-browser
```

A.3.2 Basic Test

A simple functionality test can be run using the provided script for constructing result folders and generating estimators. For example, to generate 10 estimators for the Gaussian mechanism with a sample size of 1000:

```
python3 construction_of_results_folders.py
Gaussian 1000 10
```

This command creates a subfolder with results in the `results/results_Gaussian/` *directory. Successful output includes printed confirmation of estimator generation and creation of result files.*

A.4 Evaluation workflow

This section outlines the steps to validate that the artifact is functional and reproduces the key claims in the paper.

A.4.1 Major Claims

(C1): *The PLRT estimator provides a uniformly convergent estimate of the f-DP curve across mechanisms and sample sizes. This is demonstrated by experiment (E1) and supported by results in Figures 1–3,9-10.*

(C2): *The Baybox auditor can reliably test a claimed privacy guarantee by checking for violations of f-DP. This is demonstrated in experiment (E2) and illustrated in Figure 4-5, 12.*

(C3): *The estimators and auditing procedure are applicable to real-world scenarios, such as evaluating trained DP-SGD models on CIFAR-10. This is demonstrated by experiment (E3) and supported by Figure 6-8.*

A.4.2 Experiments

(E1): *PLRT Estimator Evaluation [5 human-minutes + 1-120 compute-minutes per example, depending on the example]: Evaluate the PLRT estimator for various mechanisms (Gaussian, SGD, ...) and varying sample sizes. We recommend evaluating only one sample size, e.g. $n_1 = 10^4$.*

Preparation: Set up the environment and generate results using (e.g. Gaussian):

```
python3 construction_of_results_folders.py
<mechanism> <n1> <repetitions>
```

Execution: Open the notebook `PLRT_shade_plots.ipynb` in `results/Figure2_3_9_10/` and run the cells to compute and visualize estimated f-DP curves. For Figure 1 open `visualize_MSE.ipynb` in `results/Figure1/` and execute file.

Results: If all mechanisms and all samples size (takes a lot of time) were run, the jupyter notebook includes all estimated f-DP curves, shaded error plots, and comparisons to theoretical reference curves. These should replicate Figures 2-3,9-10.

(E2): *Baybox Auditor Test [1 human-minutes + 10 compute-minutes per Figure]: Evaluate the Baybox classifier-based auditor to test a privacy claim.*

Preparation: Use existing audit-ready folders in `results/Figure4/` and rerun the jupyter.

Execution: Open and execute `auditing_correct_mechanisms.ipynb` and/or `auditing_faulty_mechanisms.ipynb` in `results/Figure5/`, which runs the whole auditing process and checks if the privacy claim is violated.

Results: The notebook outputs an audit decision and

plots showing the estimated rejection region. This validates results like those in Figure 4 and Figure 5.

(E3): *DP-SGD Audit on CIFAR-10 [5 human-minutes + 15 compute-minutes]: Apply the PLRT estimator and Bay-box auditor to real-world DP-SGD models trained on CIFAR-10.*

Preparation: Use pretrained models in `trained_models/`, or regenerate them with: `./run_generate_sgd_models`
Please specify the NUM_TRAIN_SAMPLES (default 2) and NUM_WORKERS (also default 2) inside the file (this takes a lot of time –depending on the NUM_WORKERS and storage space).

Execution: Open notebooks in Figure6, Figure7, Figure8 folders (e.g., `lowerbound_DPSGD_different_loss.ipynb`) in the corresponding figure folder and apply the auditor or estimator on model predictions.

Results: The notebook outputs estimated or audited privacy guarantees. We replicate Figure6-8.

All experiments are expected to replicate the corresponding figures and validate the main findings of the paper. Minor deviations may occur due to randomness in sampling or non-deterministic model execution.

A.5 Notes on Reusability

This artifact is implemented in a modular and extensible way, enabling its reuse beyond the specific experiments reported in the paper.

The core estimator and auditor implementations are designed to be plug-and-play. Users can easily substitute one mechanism for another by modifying the input arguments to the estimator scripts or by generating their own model outputs. Specifically:

- New mechanisms (e.g. subsampled Laplace) can be evaluated by generating samples and formatting them in the same structure as the provided `results_mech/` folders.
- The `construction_of_results_folders.py` script allows users to easily scale up the number of estimators or change sample sizes for empirical experiments.
- The estimators are agnostic to the data distribution used; with minor preprocessing adjustments, users can audit mechanisms trained on datasets beyond CIFAR-10.

The artifact can also serve as a testing ground for new privacy estimation methods or auditing strategies by modifying or extending the implementations in the `src/` folder.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Addressing Sensitivity Distinction in Local Differential Privacy: A General Utility-Optimized Framework

Xingyu He[1], Youwen Zhu[1], Rongke Liu[1], Gaoning Pan[2], Changyu Dong[3]

[1]Nanjing University of Aeronautics and Astronautics
[2]Hangzhou Dianzi University, [3]Guangzhou University

A Artifact Appendix

A.1 Abstract

This artifact contains Python-based code designed to evaluate ULDP protocols using both real-world and synthetic datasets. The artifact includes three main components: (1) the source code of all ULDP frequency protocols evaluated in the paper (including existing mechanisms uRR, uRAP, and uHR, as well as our proposed mechanisms uSS, uUE, and uLH); (2) scripts for running the experiments; (3) a complete README file. With all these components, the results presented in our paper can be reproduced.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is publicly available on Zenodo at https://zenodo.org/records/15614307. Please use the latest version of this record.

A.2.3 Hardware dependencies

This artifact is compatible with any machine equipped with at least 16GB of RAM. GPU resources are not required to run the code; however, improvements in CPU performance and memory capacity can reduce overall runtime. We conducted our experiments on two machines: 1. i9-13980HX with 16GB RAM; 2. i7-9700K with 16GB RAM.

A.2.4 Software dependencies

This artifact was developed using Python 3.9 and should be compatible with the latest versions of Python. While we ran our experiments on Windows, Python is a cross-platform language, which means the artifact can also run on macOS and Linux. The artifact only requires three Python packages: NumPy, xxHash, and Matplotlib. All of these packages can be installed via pip. In our experiments, we used Anaconda to manage the Python environment and provide an *environment.yml* file that fully describes the environment we used.

A.2.5 Benchmarks

The three real-world datasets and one synthetic dataset used in the experiments have been integrated into the artifact.

A.3 Set-up

A.3.1 Installation

Before installation, please ensure that Python is available on your system. We recommend using Anaconda/conda, and you can reproduce our environment using the following command:

```
conda env create -f environment.yml
```

Alternatively, you may manually install the required packages for the artifact using pip:

```
pip install numpy xxhash matplotlib
```

A.3.2 Basic Test

Open the "GLUTF-Pure ULDP Experiments (without frequency item mining)\src\exp1_impact_of_epsilon.py" file in the src directory, change

```
replication = 50
```

in the four functions to

```
replication = 1
```

, and change

```
for epsilon in [0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5]:
```

in the main function at the bottom to

```
for epsilon in [0.5]:
```

Then run the file.

```
python exp1_impact_of_epsilon.py
```

If the experimental and theoretical MSE of each mechanism are successfully displayed, it means success.

A.4 Evaluation workflow

Important Tips: To improve result stability, each experiment in our code is executed 50 times by default, and the average MSE is reported. While this setting ensures reliable results, it can be time-consuming. To speed up the evaluation, we recommend setting the number of repetitions (the corresponding parameters in the code are *replication* or *loop*) to 5 or 10 during initial testing. If the observed results deviate significantly from the theoretical MSEs due to randomness, increasing the replication count can help stabilize the outcomes.

A.4.1 Major Claims

(C1): *The experimental MSE of each mechanism decreases as the epsilon increases. Moreover, uSS is definitely better than uRR, and uUE is definitely better than uRAP. The utility of uLH is almost the same as that of uUE. This is proven by the experiment (E1) described in Section 6.2 whose results are illustrated in Figure 4.*

(C2): *The experimental MSE of each mechanism increases as the sensitive ratio increases. This is proven by the experiment (E2) described in Section 6.2 whose results are illustrated in Figure 5.*

(C3): *The MSE of the mechanism using collaborative sampling is almost the same as the MSE of the mechanism using the true theta. This is proven by the experiment (E3) described in Section 6.3 whose results are illustrated in Figure 6.*

(C4): *When we reduce z to 0, the mse will decrease in most cases (if the maximum value of z is 0, this action will have no effect). This is proven by the experiment (E4) described in Section 6.4 whose results are illustrated in Figure 7.*

(C5): *The experimental MAE of each mechanism decreases as the epsilon increases. Moreover, uSS is definitely better than uRR, and the utility of uLH is almost the same as that of uUE. This is proven by the experiment (E5) described in Section 6.5 whose results are illustrated in Figure 8.*

(C6): *Both the F1 and NDCG of the mechanisms increase with the increase of privacy budget. uRR performs the worst, while the other mechanisms perform similarly. This is proven by the experiment (E6) described in Section 6.5 whose results are illustrated in Figure 9.*

A.4.2 Experiments

(E1): *[exp1_impact_of_epsilon.py] [1.6 * replication (default is 50) compute-hour]: This experiment aims to evaluate the impact of privacy budget on the MSE of the ULDP mechanism.*
Preparation: *For faster verification, users can customize the parameter "replication" to a smaller number (such as 5 or 10), but the results will fluctuate more.*
Execution: *Enter the GLUTF-Pure ULDP Experiments (without frequency item mining)\src directory and execute:*

```
python exp1_impact_of_epsilon.py
```

Results: *The MSE of the ULDP mechanism will be output to the console. You are free to plot the results. You can feed the MSE into the exp_1_all_exp function in experiment_visualization.py to generate the graph, or you can use any method you like.*

(E2): *[exp1_impact_of_sensitive.py] [1.6 * replication (default is 50) compute-hour]: This experiment aims to evaluate the impact of sensitive ratio on the MSE of the ULDP mechanism.*
Preparation: *For faster verification, users can customize the parameter "replication" to a smaller number (such as 5 or 10), but the results will fluctuate more.*
Execution: *Enter the GLUTF-Pure ULDP Experiments (without frequency item mining)\src directory and execute:*

```
python exp1_impact_of_sensitive.py
```

Results: *The MSE of the ULDP mechanism will be output to the console. You are free to plot the results. You can feed the MSE into the exp_1_vary_xs function in experiment_visualization.py to generate the graph, or you can use any method you like.*

(E3): *[exp2_collaborative_sampling.py] [0.8 * replication (default is 50) compute-hour]: This experiment aims to evaluate whether the use of cooperative sampling has an impact on the MSE of the ULDP mechanism.*
Preparation: *For faster verification, users can customize the parameter "replication" to a smaller number (such as 5 or 10), but the results will fluctuate more.*
Execution: *Enter the GLUTF-Pure ULDP Experiments (without frequency item mining)\src directory and execute:*

```
python exp2_collaborative_sampling.py
```

Results: *The MSE of the ULDP mechanism with collaborative sampling will be output to the console. The MSE of the ULDP mechanism with true theta can be found in E1. You are free to plot the results. You can feed the MSE into the exp_2_uSS, exp_2_uOUE and exp_2_uOLH function in experiment_visualization.py to generate the graph, or you can use any method you like.*

(E4): *[exp3_without_z.py] [0.7 * replication (default is 50) compute-hour]: This experiment aims to evaluate the impact of z on the MSE of the ULDP mechanism.*
Preparation: *Before executing this file, the protocol code needs to be modified. Specifically, when initializing the definition parameters, add the following code at the end of the __init__ function:*

```
self.z = 0
self.z_star = 1-self.f
```

For faster verification, users can customize the parameter "replication" to a smaller number (such as 5 or 10), but the results will fluctuate more.
Execution: *Enter the GLUTF-Pure ULDP Experiments (without frequency item mining)\src directory and execute:*

```
python exp3_without_z.py
```

Results: *The MSE of the ULDP mechanism will be output to the console. You are free to plot the results. You can feed the MSE into the exp_3_without_z_uSS, exp_3_without_z_uUE and exp_3_without_z_uLH function in experiment_visualization.py to generate the graph, or you can use any method you like.*

(E5): *[exp4_mae.py] [1.6 * replication (default is 50) compute-hour]: This experiment aims to evaluate the impact of privacy budget on the MAE of the ULDP mechanism.*
Preparation: *For faster verification, users can customize the parameter "replication" to a smaller number (such as 5 or 10), but the results will fluctuate more.*
Execution: *Enter the GLUTF-Pure ULDP Experiments (without frequency item mining)\src directory and execute:*

```
python exp4_mae.py
```

Results: *The MAE of the ULDP mechanism will be output to the console. You are free to plot the results. You can feed the MSE into the exp_4_mae function in experiment_visualization.py to generate the graph, or you can use any method you like.*

(E6): *[Folder:(only for frequency item mining)] [0.16 * replication (default is 50) compute-hour]: This experiment aims to evaluate the impact of privacy budget on the F1 and NDCG of the ULDP mechanism.*
Preparation: *For faster verification, users can customize the parameter "loop" to a smaller number (such as 5 or 10), but the results will fluctuate more.*
Execution: *Enter the GLUTF-Pure ULDP Experiments (only for frequency item mining)\src directory and execute:*

```
python exp-<mechanism>-<dataset>.py
```

Results: *The results will be stored in the results folder in txt format. You are free to plot the results. You can feed the result into the exp_4_item_mining function in experiment_visualization.py to generate the graph, or you can use any method you like.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Further Study on Frequency Estimation under Local Differential Privacy

Huiyu Fang
Southeast University

Liquan Chen
Southeast University

Suhui Liu
Southeast University

A Artifact Appendix

A.1 Abstract

The artifact consists of Python-based code and scripts designed to evaluate LDP frequency protocols using synthetic and real-world datasets. The artifact comprises three main components: (1) the source code of all LDP frequency protocols evaluated in the paper, including the baseline and proposed protocols, (2) the scripts to conduct the experiments and plot the results, and (3) a comprehensive README file. With all these components, the results of our paper can be reproduced.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is publicly available on Zenodo at https://zenodo.org/records/14715748. Please use the latest version of this record.

A.2.3 Hardware dependencies

The artifact is compatible with any commodity computer with a minimum of 16GB of RAM and requires approximately 300MB of disk space. The memory requirement mainly comes from the runtime experiment and more RAM is required if the customized domain size in the runtime experiment is larger than 4096. Multi-core CPUs can significantly speed up the experiments. We run the experiments on a PC with AMD Ryzen 9 7950X and 64GB memory.

A.2.4 Software dependencies

The artifact is developed in Python 3.10.4 and should be compatible with recent Python versions. We run the experiments on Windows, but Python is a cross-platform language, which means it can run on any OS with Python installed, including Windows, macOS, and Linux. The artifact requires only four Python packages, including NumPy, xxHash, PyArrow, and Matplotlib. All these packages can be installed from PyPI using pip. The *requirements.txt* file is provided for convenient installation.

A.2.5 Benchmarks

The real-world data set is from the NYC Taxi and Limousine Commission and is integrated into the artifact.

A.3 Set-up

A.3.1 Installation

Before installation, a recent Python environment should be created. We recommend Python 3.10.4. After downloading and unzipping the artifact, enter the artifact folder and run the following command in the terminal:

```
pip install -r requirements.txt
```

The command will install all dependencies required by the artifact.

A.3.2 Basic Test

Once the installation is completed, users can run the following command in the terminal to verify that all required software components are functioning properly:

```
python test.py
```

The script will run all unit tests and print a success message in the terminal after passing each unit test.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The empirical MSE matches our analytical MSE of all frequency protocols in the paper. This is proven by the experiment (E1) described in Section 5.2 whose results are illustrated in Figure 4.*

(C2): *RUE and RLH achieve better MSE or accuracy than OUE and OLH, especially for the small domain and top frequent values. RWS achieves the same optimal accuracy as SS but with less communication costs for large domains. This is proven by the experiments (E1) described in Section 5.3 whose results are illustrated in Figures 5 and 6.*

(C3): *The aggregation time of RLH and RWS are much smaller than that of OLH for large domains. This is proven by the experiments (E2) described in Section 5.4 whose results are illustrated in Figure 7.*

A.4.2 Experiments

We give complete instructions for users to run all experiments and plot all experimental results in the paper in the README file. More details, especially the experimental parameters, can be found in the README file.

(E1): *main.py [approximately 105 compute-hour + 250MB disk]: This experiment is to achieve the empirical MSE of all frequency protocols on both synthetic and real-world datasets with a specified range of* ε *and domain size.*

Preparation: We default to running the experiment 100 times to take the average MSE but very time-consuming. For quick verification, users can customize the parameter *run_repeat_time* in the *main.py* and *drawTopk.py* files to a small number (e.g. 10), but the results for small domains would fluctuate more. The ranges of *run_d_range* and *run_epsilon_range* can also be reduced.

Execution: Enter the artifact folder and run the following command in the terminal:

```
python main.py
```

Experimental results will be saved in the *results* folder. More specifically, each estimated frequency result will be saved in the *results/protocol_name* folder and the summary of average empirical MSE results will be saved in CSV files.

Results: After the main experiment, plot Figures 4, 5, and 6 in the paper by running:

```
python drawAnaEmpMSE.py
python drawEmpMSE.py
python drawTopk.py
```

The plotted figures will be saved in the *draw* folder.

(E2): *runtime.py [approximately 6 compute-hour + 5.5GB RAM]: This experiment is to demonstrate the aggregation time of all frequency protocols in the paper.*

Preparation: Runtime and memory costs are linearly related to data size, so the taxi dataset requires approximately 36 times as much runtime and RAM as the synthetic dataset. For quick verification or if memory is limited, users can run the experiment only on the synthetic dataset by commenting out lines from 156 to 158 of the *runtime.py* code and lines from 66 to 79 of the *drawRuntime.py* code.

Execution: Enter the artifact folder and run the following command in the terminal:

```
python runtime.py
```

The summary of runtime results will be saved in CSV files in the *results* folder.

Results: After the runtime experiment, plot Figures 7 in the paper by running:

```
python drawRuntime.py
```

The plotted figure will be saved in the *draw* folder. Additionally, Figure 5, 6, and 7 share the same legend, which is generated by *drawEmpMSE.py* and saved as *legend.eps* in the *draw* folder.

A.5 Notes on Reusability

The source code of frequency protocols is modularized for reusability. The Python-based scripts of experiments can be used as examples for users to conduct experiments on different datasets. However, to be more practical, the source code may need to be divided into client-side and server-side. More specifically, the perturb function should be implemented on the client side, while the aggregate function should be implemented on the server side.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Stack Overflow Meets Replication: Security Research Amid Evolving Code Snippets

Alfusainey Jallow, Sven Bugiel

A Artifact Appendix

A.1 Abstract

The study examines the impact of Stack Overflow code evolution on the stability of prior research findings derived from Stack Overflow data and provide recommendations for future studies. We systematically reviewed papers published between 2005–2023 to identify key aspects of Stack Overflow that can affect study results, such as the language or context of code snippets. Our analysis reveals that certain aspects are non-stationary over time, which could lead to different conclusions if experiments are repeated at different times. We replicated six studies using a more recent dataset to demonstrate this risk. Our findings show that four papers produced significantly different results than the original findings, preventing the same conclusions from being drawn with a newer dataset version. The code and data artifact are permanently hosted on Zenodo, allowing verification of the claims made in this study.

A.2 Description & Requirements

Our artifact provides the necessary components to reproduce the experiments and results presented in our paper. It consists of two main parts: the source artifact, which includes the code and environment setup, and the data artifact, which contains the datasets used in our experiments.

To ensure reproducibility, we have included a *requirements.txt* file that lists all Python dependencies used in the project. Evaluators should use this file to create a conda environment, ensuring the exact setup used for the experiments. The source artifact is stored on GitLab, and the data artifact is available on Zenodo.

For reproducibility, evaluators should download the source code and data from Zenodo. The artifact has been designed to be run without any security, privacy, or ethical concerns, and we have ensured that the setup is consistent and reliable for evaluation.

A.2.1 Security, privacy, and ethical concerns

Our artifact does not pose any security for the evaluators. There are no destructive steps involved in its execution, nor are any security mechanisms disabled during its operation.

A.2.2 How to access

The complete artifact for this paper is available on Zenodo at the following URL: `https://zenodo.org/records/14733196`. Download the archive file and extract its contents to a local directory.

A.2.3 Hardware dependencies

To run the scripts in the *sources/**/analysis* directory, we tested on a machine with 756 GB of RAM and a CPU with 64 cores, along with 3.5 TB of disc space. However, we did not use all the 3.5 TB space and we believe that a disc space of 1.5TB would also work. However, to reproduce the major claims described in Section A.4.1, you only need to run the scripts in the `sources/**/analysis` directory. The instructions for running each experiment to reproduce the major claims are described in Section A.4.2.

A.2.4 Software dependencies

Our artifact relies on the following software dependencies:

- **virtual environment**: A software for creating python virtual environment, e.g., Conda or Venv, is required.
- **Docker** for creating the MySQL docker container.
- **7-Zip** for extracting the artifact archive file stored on Zenodo.

A.2.5 Benchmarks

We have released the data set required for the experiments reported in our paper on Zenodo. The data set can be accessed and downloaded from the following stable URL: `https://zenodo.org/records/14733196`.

A.3 Set-up & Installation

Create a fresh virtual environment and install the required packages using *pip*. The example below shows how to create a new conda environment named 'usenix-ae' and install the required packages:

```
$ conda create -n usenix-ae python=3.12
$ conda activate usenix-ae
$ pip install -r requirements.txt
```

Afterward, set up a Docker container with a MySQL server, according to the following steps, that holds the `sotorrent22` database (and the required tables) needed to replicate claims **C1**, **C7**, **C13**, **C14**, **C15**, and **C16** described in Section A.4.1.

```
1. cd into the directory
cd mysql-docker

2. Build the Docker image
$ docker build -t paper403 .

2. Run the container (in detached mode)
$ docker run --name mysql-server -d \
  -p 3306:3306 paper403

3. Connect to the DB:
$ docker exec -it mysql-server mysql \
  -u sotorrent -psotorrent
```

Finally, follow the instructions provided in Step 5 of the Zenodo description to import the necessary database tables.

A.3.1 Basic Test

Once the MySQL server docker container is running, connect to the server and list all the databases on the server. Confirm that the *sotorrent22* database is listed:

```
1. Connect to the server
docker exec -it mysql-server mysql -u sotorrent -p

2. List the databases. Confirm that sotorrent22 is listed
$ show databases

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| performance_schema |
| sotorrent22        |
+--------------------+
3 rows in set (0.01 sec)
```

Finally, confirm that all the database tables have been successfully imported into the *sotorrent22* database:

```
mysql> use sotorrent22
mysql> show tables;

+--------------------+
| Tables_in_sotorrent22 |
+--------------------+
| Posts              |
| PostHistory        |
| Users              |
| CppCheckWeakness   |
| CrawledPapers      |
| DicosResults       |
| Violations         |
| PostReferenceGH    |
+--------------------+
```

A.4 Evaluation workflow

Below, we outline the key claims in the paper along with the corresponding experiments to reproduce each claim. The evaluators identified minor discrepancies for **C9** (observed: 30,359, expected: 30,343) and **C18**: *no-dupe-keys* under **Parse Errors (PE)** violations (observed 874 vs. expected 2,070). These discrepancies have been addressed in the extended version of the paper, which is available on Arxiv[1].

A.4.1 Major Claims

(C1, Optional): *Our literature search yielded 42 relevant studies. This is proven by the experiment (E1) described in Section 3.2 whose results are reported in Table 1.*

(C2): *Certain programming languages have exhibited distinct trends regarding the overall number of added snippets and the ratio of security-relevant edits, with languages like C/C++ being relatively stable, while others like JavaScript and Python have fluctuated over time. This claim is demonstrated by the monthly trends in the addition of new code snippets shown in **Figure 3** and the analysis of post edits in **Figure 4** of Section 5.*

(C3): *The ratio of security-relevant comments on Stack Overflow has steadily increased, suggesting an ongoing community effort to improve the quality of shared content. This claim is backed by the data shown in **Figure 5**, highlighting the increasing percentage of security-relevant commits and comments over time.*

(C4, page 8): *Code snippets with weaknesses (i.e., $Code_w$) increased (11,748 ↗ 30,254), indicating a growth rate of 157.5%.*

(C5, page 8): *The authors identified 12,998 CWE instances in the latest versions of 7,481 answers, whereas we found 7,679 instances in 5,721 answers.*

(C6, page 8): *We noticed a shift in the ranking of CWE types: CWE-758 climbed 6th ↗ 2nd; CWE-401 dropped 2nd ↘ 3rd; CWE-775 fell 3rd ↘ 7th.*

(C7, page 8): *As revisions increased from 1 to 3 or more, the proportion of improved $Code_w$ only rose from 3.1% to 7.4%. This claim is shown on the right-hand side of Table 7 in Appendix A.*

[1] https://arxiv.org/abs/2501.16948

(C8, page 8 and page 9): *We found that the fraction of users with vulnerable C/C++ snippets more than doubled compared to the original findings. Moreover, the number of users that contributed just one vulnerable snippet also increased. Further, the authors reported that users who contributed multiple vulnerable snippet versions repeatedly contributed the same CWE type. We found that these users contribute different types of CWE with the same likelihood.*

(C9, page 10): *Using the same approach on our dataset of 1,046,052 posts, we discovered 30,359 (2.9%) insecure posts, i.e., an* ***increase of 52%****. Among these, 4,887 (16.1%) insecure posts contained all three features, i.e., an* ***increase of 155%****, while the remaining 25,472 posts contained two features.*

(C10, page 10): *We found that* DICOS *had an 11% precision (compared to the authors' 91%), 32% accuracy (versus 89%), and an 87% recall (versus 93%). These results are shown in Table 4. This low precision and accuracy indicates that the ability of* DICOS *to detect insecure code snippets has decreased significantly.*

(C11, page 11): *Using the newer version of the SOTorrent dataset, we observed a higher ratio of insecure posts between accepted (7.72%) and non-accepted (6.61%) answers. Figure 10 in Appendix B compares the original and replication results.*

(C12, page 11): *The types of security weaknesses detected in C/C++ code snippets have changed over time, with memory leaks becoming the most prevalent, replacing undefined behavior as the dominant issue. This claim is supported by the comparison of the original and replication results for RQ3, as seen in Section 6.2.3. Figure 11 in Appendix B compares the original and replication results.*

(C13, page 12): *After applying the authors' filtering criteria, we obtained 12,095 questions containing 72,202 answers, of which 10,140 were accepted answers. This means the number of code snippets matching the authors' filtering criteria has dropped since 2018: 529,054 ↘ 239,575.*

(C14, page 12): *Our findings regarding the number of questions with at least one insecure answer differ significantly: 18.1% (out of 10,861) dropped to 4.9% (out of 12,095). Similarly, the percentage of accepted answers containing at least one insecure snippet also decreased: 9.8% (out of 7,444) ↘ 2.2% (out of 10,139).*

(C15, page 12): *Although the vulnerability rankings from the original study remain unchanged, we observed a shift in the number of affected snippets: code injection increased (2,319 ↗ 5,734), while insecure cipher (564 ↘ 356), insecure connection (624 ↘ 276), and data serialization (153 ↘ 140) all dropped. Like the original study, no snippets were impacted by XSS vulnerabilities.*

(C16, page 12): *Like the original study, we also found no significant difference between the two user groups. However, we observed different $p-value$ (0.9 ↗ 6.2) and Cliff's delta (0.01 ↗ 0.03) values.*

(C17, page 13): *The number of violations in JavaScript code snippets increased from 5,587,357 to 7,385,044, with the average violations per snippet rising from 11.94 to 28.8. This claim is supported by the results in Section 6.4.3 (****Revisiting RQ1 Findings****).*

(C18, page 13): *Due to Stack Overflow's evolution, the number of violations in each category has risen. Further, the ranking of violations within certain categories has changed. This claim is supported by the results in Section 6.4.3 (****Revisiting RQ2 Findings****).*

A.4.2 Experiments

For all the experiments described below, make sure to create and activate a Python virtual environment based on the *requirements.txt* file (see Section A.3). All the experiments require the virtual environment to be set up and activated. Not all of the experiments listed below require the MySQL server Docker container to be running. The claims that do require the Docker container are **C1**, **C7**, **C13**, **C14**, **C15**, and **C16**. The other claims do not require the container for verification.

(E1 (Optional)): *[Table 1, Section 3.2] [10 mins human-hour + ≈20 mins compute-time]:* The goal of this experiment is to reproduce **C1**, which pertains to the number of research studies identified through our systematic literature review process outlined in Figure 2. The relevant studies discovered are presented in Table 1, and the aim of this experiment is to replicate the studies listed in Table 1.

Preparation: *Ensure that the MySQL Server Docker container is running and that you can connect to the server as outlined in Section A.3.1. You will also need access to an OPENAI API Key. The script requires the API Key and the* `DB_HOST` *to be set in the environment where it is being executed. For example, when running the script from the terminal, the* `OPENAI_API_KEY` *should be configured in the environment. If the default database user and password created during the creation of the docker container is not changed, then the script will use those credentials to connect to the DB server on localhost. Otherwise, configure the* `DB_HOST` *environment variable with the server's IP address.*

Execution: *Run* `python relevant_studies.py`

Results: *The script will display all the relevant studies described in Table 1.*

(E2): *[Section 5] [10 mins human-hour + ≈15 mins compute-time]: The goal of this experiment is to reproduce claims* ***C2*** *and* ***C3****, which pertains to the statistics about the evolution of code and their surrounding context (i.e., comments and commit messages) on Stack Overflow.*

The claims and statistics supporting them are described in Section 5.

Preparation: *Ensure that the virtual environment is activated and contains all the packages listed in the requirements.txt file. Also ensure that the **data** directory is located in its proper location and contains all necessary files. All related scripts for this experiment are located in the* `sources/5_evolution_of_stackoverflow` *directory.*

Execution: *Run all the code cells in the* `Evolution.ipynb` *and* `PCS.ipynb` *Jupyter notebooks, which is found in the* `5_evolution_of_stackoverflow` *folder. The notebooks include direct quotes from the paper in markdown cells, with the corresponding code to reproduce each quote provided right after it.*

Results: *Each code cell in the notebook, once executed, will reproduce the claims C2 and C3 described in Section 5 of the paper.*

(E3): *[Section 6.1] [10 mins human-hour + ≈20 mins compute-time]: The goal of this experiment is to demonstrate how claims C4, C5, C6, C7, and C8 described in Section 6.1 can be reproduced.*

Preparation: *Ensure that the virtual environment is activated and contains all the packages listed in the requirements.txt file. The* `table_7.py` *script connects and queries the sotorrent22 database running in the MySQL Server docker container. All related scripts for this experiment are located in the* `case_study_1/analysis` *directory.*

Execution: *Execute the following scripts to reproduce the findings for each claim.*

C4: *Execute the* `main()` *function inside the* `table_2.ipynb` *Jupyter notebook.*

(C5, C6): *Execute all code cells inside the* `RQ1_RQ2.ipynb` *Jupyter notebook.*

C7 *Run* `python table_7.py`

C8 *Study and execute all code cells of the* `Zhang.ipynb` *Jupyter notebook. The notebook includes direct quotes from the paper, with the corresponding code to reproduce each quote provided right after it.*

Results: *Executing the scripts described above will reproduce claims C4, C5, C6, C7 and C8.*

(E4): *[Section 6.2] [10 mins human-hour + ≈20 mins compute-time]: The goal of this experiment is to demonstrate how claims C9, C10, C11 and C12 described in Section 6.2 can be reproduced.*

Preparation: *Ensure that the virtual environment is activated and contains all the packages listed in the requirements.txt file. All related scripts for this experiment are located in the* `case_study_2/analysis` *directory. The results for Claim C10 were obtained through manual labeling conducted by two researchers. The* `data/dicos_discovery_accuracy.xlsx` *CSV file includes multiple tabs, each containing the coding label results for each task. The **Discovery Accuracy** tab includes the results for the precision, accuracy, and recall calculations.*

Execution:

(C9, C11, C12): *Run all the code cells in the* `main.ipynb` *Jupyter notebook, which is found in the* `case_study_2/analysis` *directory. The notebook includes direct quotes from the paper in markdown cells, with the corresponding code to reproduce each quote provided right after it.*

(C10): There is no Python script to run in order to verify this claim. Instead, the results of this experiment can be found in the Discovery Accuracy tab of the `data/dicos_discovery_accuracy.xlsx` CSV file.

Results: *Each execution of the python script will reproduce the findings for each claim.*

(E5): *[Section 6.3] [10 mins human-hour + ≈10 mins compute-time]: The goal of this experiment is to demonstrate how claims C13, C14, C15, and C16 described on page 12 in Section 6.4 can be reproduced.*

Preparation: *This experiment utilizes scripts that require DB server credentials to connect to and query the sotorrent22 database. The scripts expect the environment variables* `DB_USER` *and* `DB_PASSWORD` *to be set, containing the database username and password. These variables will be used by the script to connect to the MySQL Server Docker container, with the default host set to localhost (127.0.0.1). All related scripts for this experiment are located in the directory* `case_study_3/analysis`*.*

Execution: *Execute the following scripts to reproduce the findings for each claim.*

C13: *Execute the code cells in* `general_stats.ipynb`

C14: `python revisiting_RQ1_findings.py insecure_post_stats`

C15 `python revisiting_RQ1_findings.py rankings`

C16 `python revisiting_RQ2_findings.py`

Results: *Each execution of the python script will reproduce the findings for each claim.*

(E6): *[Section 6.4] [10 mins human-hour + ≈10 mins compute-time]: The goal of this experiment is to reproduce claims **C17** and **C18**, described on page 12 in Section 6.4.*

Preparation: *Ensure that the virtual environment is activated and contains all the packages*

listed in the requirements.txt file. This experiment uses the `violations.feather` *and* `rule_categories.feather` *files located in the* `data/feather_files` *directory.*
Execution: *Run all the code cells in the* `main.ipynb` *Jupyter notebook, which is found in the* `case_study_4/analysis` *directory. The notebook includes direct quotes from the paper in markdown cells, with the corresponding code to reproduce each quote provided right after it.*
Results: *Each code cell in the notebook, once executed, will reproduce claims C17 and C18.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Beyond Exploit Scanning: A Functional Change-Driven Approach to Remote Software Version Identification

Jinsong Chen[†*], Mengying Wu[†*], Geng Hong[†*], Baichao An[†], Mingxuan Liu[‡], Lei Zhang[†], Baojun Liu[§], Haixin Duan[§¶] and Min Yang[†]
[†]Fudan University, {jschen23, wumy21}@m.fudan.edu.cn, {ghong, bcan20, zxl, m_yang}@fudan.edu.cn
[‡]Zhongguancun Laboratory, liumx@mail.zgclab.edu.cn
[§]Tsinghua University, {lbj, duanhx}@tsinghua.edu.cn
[¶]Quancheng Laboratory

A Artifact Appendix

A.1 Abstract

In this paper, we present *VersionSeek*, a new functionality-based tool for software version identification through functional probe generation and automated response analysis. It includes the source code for software deployment, functional probe generation, response processing, and version identification. The artifact also contains ethically filtered and validated probes along with their corresponding locally generated response outputs. In addition, it provides the source code and training data used to construct the comparative baseline. A comprehensive README file is included to facilitate the effective use of the version identification tool.

It provides capabilities for functional probe generation, automated deployment, and version identification for five remote software systems. Additionally, validation experiments are included to demonstrate the effectiveness of these capabilities.

A.2 Description & Requirements

In this section, we detail the accessibility of our artifact and specify the required hardware and software environments.

Our artifact consists of seven main folders: Deployment, Generation, ResponseProcessing, VersionIdentification, Probes, ProbeOutputs, and Comparison. The `Deployment` folder contains automated deployment code for five open-source software systems. The `Generation` folder includes code for automatically generating functional probes using large language models (LLMs), guided by functional changes and their corresponding contextual information. The `ResponseProcessing` folder implements response standardization and classification modules. The `VersionIdentification` folder provides automated version identification modules tailored to the selected software targets. The `Probes` and `ProbeOutputs` folders contain ethically filtered and validated functional probes, along with their locally generated response outputs. Finally, the `Comparison` folder provides baseline implementations and training data for various machine learning and black-box approaches used in our comparative evaluation.

A.2.1 Security, privacy, and ethical concerns

We performed a comprehensive assessment of both the codebase and datasets used in this study to ensure that no personally identifiable information (PII) or sensitive content was misused or exposed. In particular, since public access to response data or identification results derived from real-world servers could enable malicious exploitation or targeted attacks, such data is not released as part of the artifact. Accordingly, this artifact does not pose any additional ethical concerns. Moreover, the artifact does not involve any destructive steps during its execution.

A.2.2 How to access

Our artifact is available through Zenodo. The artifact can be accessed at https://doi.org/10.5281/zenodo.15576928.

A.2.3 Hardware dependencies

We implemented this artifact on a server with a 13th Gen Intel® Core™ i7-13700 processor (24 cores), 32GB of memory, and a 1TB SSD. Additionally, the LLM used in our experiments, Qwen2.5-32B-Instruct, was deployed on a server equipped with an Intel® Xeon® Gold 6330 CPU @ 2.00GHz

[1]These authors contributed equally to this work.

(28 cores), 8 NVIDIA GeForce RTX 4090 GPUs, 512 GB of RAM, 29 TB of HDD storage, and 447 GB of SSD storage.

A.2.4 Software dependencies

Although our experiments were conducted primarily on Ubuntu 23.04, the artifact is compatible with any Linux-based environment that supports Python 3.11. However, we recommend using Ubuntu 22.04 or Ubuntu 24.04 for better stability. We use Miniconda for environment and dependency management, and all required packages are specified in the requirements.txt file located at the root of the repository.

A.2.5 Benchmarks

None.

A.3 Set-up

This section provides detailed instructions for setting up our experimental environment. The setup utilizes MiniConda for environment management and includes a verification step to confirm the correct installation of all components.

A.3.1 Installation

To ensure full compatibility with *VersionSeek*, we recommend creating a new conda environment with Python 3.11:

```
conda create -n versionseek python=3.11
conda activate versionseek
```

Next, install the required dependencies:

```
pip install -r requirements.txt
cd Generation
pip install -e ./"[gui,rag,code_interpreter,mcp]"
```

In addition, Docker and docker-compose are required for deployment. Installation instructions can be found at:

- https://docs.docker.com/engine/install/
- https://docs.docker.com/compose/install/standalone/

The necessary `redis-tools` package can be installed using:

```
sudo apt update
sudo apt install redis-tools
```

To ensure the proper functioning of certain services (e.g., Elasticsearch), the following system parameter must also be configured:

```
sudo sysctl -w vm.max_map_count=262144
```

A.3.2 Basic Test

We provide a simple command to verify whether all required system commands and Python dependencies are properly installed:

```
python versionSeek.py --test
```

Upon successful verification, the output will be: *All required commands and Python modules are available.*

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *VersionSeek* has been utilized to automatically generate new probes based on functional feature descriptions. This is proven by the experiments (E1).
(C2): *VersionSeek* provides automated deployment capabilities for five different software versions. This is proven by the experiments (E2).
(C2): *VersionSeek* provides automated identification capabilities for five remote software versions, as demonstrated in Experiment (E3).

A.4.2 Experiments

This section outlines the experimental procedures used to validate our key claims. Each experiment is organized into three main blocks: Preparation, Execution, and Results.

(E1): [Functional Probe Generation] [15 human-minutes + 20 compute-minutesr + 5GB disk]: This experiment demonstrates the capability to automatically generate probes using the wildcard feature introduced in Elasticsearch 7.9.1 as a representative example.
Preparation: The following steps should be performed within the `Generation` directory. First, the `MODEL_NAME`, `MODEL_URL`, and `API_KEY` fields must be specified in the `probeGeneration.py` file. We **strongly recommend** using `Qwen2.5-32B-Instruct` as the underlying model. The `MODEL_URL` typically takes the form `http://host/v1`, where `host` refers to the address of the server hosting the LLM deployment, which may either be a locally deployed model or a third-party API service such as OpenRouter.ai.
Execution: Run the following command to start the probe generation process:
`python probeGeneration.py`
Results: After the script runs successfully, it automatically generates two files in the ./tmp directory. The filenames follow the pattern timestamp_search_releate_filepath.json and timestamp_get_interact_command.json, which store the list of the most relevant retrieved documents and the generated probes, respectively.

(E2): [Deployment] [20 human-minutes + 20 compute-minutes]: This experiment illustrates the capability of automated deployment by deploying a single version across five different software applications.
Preparation: Following the procedures outlined in Section A.3.1, complete the deployment of Docker and docker-compose, and ensure that the network supports pulling images from Docker Hub.
Execution: Run the following command to start the deployment process:
`python versionSeek.py --test-deploy`
Results: The terminal will display the installation logs along with the results of the commands used to verify the success of the installation.

(E3): [Version Identification] [30 human-minutes + 30 compute-minutes]: Building upon the software deployed in Experiment E2, this experiment identifies their corresponding versions, demonstrating the capability of automated remote software version identification.
Preparation: Following the procedures outlined in Section A.3.1, complete the installation of redis-tools and configure the vm.max_map_count parameter accordingly.
Execution: Run the following command to initiate the version identification process:
`python versionSeek.py --test-identify`
Results: The terminal will display the installation logs along with the version identification results, highlighted in green. These include the reason for termination, matched probing paths, and the most likely inferred version.

A.5 Notes on Reusability

A.5.1 Debugging Support

To assist users in diagnosing service-related issues, we outline a recommended procedure for debugging both the deployment and version identification functionalities of `VersionSeek`.

For example, to test whether a specific version of Joomla (e.g., version 5.1.0) is successfully deployed, users can run the following command:

```
python versionSeek.py -d --software joomla \
  --port 8880 --version 5.1.0
```

Concurrently, users can check if it is accessible by opening another terminal and executing:

```
nc -zv -w 3 127.0.0.1 8880
```

A successful connection indicates that the deployment has completed and the service is reachable.

While the service is still running, users can validate the version identification functionality by executing the following command in a separate terminal:

```
python versionSeek.py -i --software joomla \
--host 127.0.0.1 --port 8880
```

This command queries the running instance and attempts to infer the deployed software version.

A.5.2 Applicability Support

Additionally, *VersionSeek* provides scripts that assist users in performing semi-automated operations to add support for new software versions. The typical workflow is as follows:

1. Follow the `README` in the **Generation** directory to fill in the required parameters and generate new probes.
2. Use these newly generated probes to invoke the **Deployment** module, which collects responses from the target software.
3. Process the collected responses by calling the methods provided in the **ResponseProcessing** directory.
4. Save the final valid probes in the **Probes** folder in the specified format, and store the corresponding responses in the **ProbeOutput** directory.
5. Modify `tree2scan` and `versionidentification` in the **VersionIdentification** module according to the generated probes to add support for the new software versions.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix "I'm trying to learn... and I'm shooting myself in the foot": Beginners' Struggles When Solving Binary Exploitation Exercises

James Mattei*, Christopher Pellegrini*, Matthew Soto*
Marina Sanusi Bohuk†, and Daniel Votipka*
*Tufts University; †MetaCTF

A Artifact Appendix

Welcome to our artifact. There are various files provided in the OSF repository; the three that are directly needed for evaluating functionality and reproducibility are Artifact.zip, CleanData.zip, and Artifact.py.

A.1 Abstract

[Mandatory] Vulnerability discovery is an essential security skill that is daunting for beginners to develop as there is a large body of knowledge to learn. They are often encouraged to search the web for information or "Just Google It!", but whether this advice is effective is unclear. We conducted semi-structured observational interviews with 37 vulnerability discovery beginners attempting to exploit 51 vulnerable programs. We capture the questions beginners have when trying to identify and exploit vulnerabilities, how they search for answers, and the challenges they face applying the results of their searches. We performed a rigorous qualitative coding of our dataset of 3950 events characterizing our participants' actions to identify several behaviors and obstacles they faced, along with quantitative measures to determine their most frequent issues.

We found beginners struggle to understand how to exploit vulnerabilities, craft their solutions, and even complete common technical tasks. They were often unable to find relevant information online to overcome these struggles, as they lacked the relevant vocabulary to craft effective keyword searches. When they did find relevant web pages, they struggled to properly transfer information from the web to their challenges due to misunderstandings and missing context. Our artifact contains the raw data for each participant run, along with the code to produce the key statistics and figures in the paper. The class definitions and data model for our participant data are flexible and allow for further analysis or use of the data set. Please take note of the chi-squared statistical tests, the generated line graph, and the counts of different events.

A.2 Description & Requirements

[Mandatory] To properly run the artifact you will need to extract both CleanData.zip and Artifact.zip, placing those folders at the same location as Artifact.py. You will need a valid installation of Python 3 and approximately 2 MB of storage for the data.

A.2.1 Security, privacy, and ethical concerns

[Mandatory] There should be minimal to no security, privacy, or ethical concerns for evaluators of our artifact. The data in CleanData.zip is already anonymized and only has the timestamps of our applied codebook. It does not contain any raw interview videos or materials that could be used to deidentify participants. Similarly, our artifact is not inherently destructive, so there are no risks to running Artifact.py.

A.2.2 How to access

[Mandatory] All artifact components can be found at the following OSF repository `https://osf.io/y6xew/`. Not all files at this repo need to be downloaded for evaluating the functionality and reproducability; only Artifact.zip, CleanData.zip, and Artifact.py are needed to run the artifact properly.

A.2.3 Hardware dependencies

[Mandatory] None

A.2.4 Software dependencies

[Mandatory] None

A.2.5 Benchmarks

[Mandatory] None

A.3 Set-up

[Mandatory]

A.3.1 Installation

[Mandatory] First, you must extract the two zip files Artifact.zip and CleanData.zip so that their contents are stored at the same location as Artifact.py. Next, you must activate the Artifact Python virtual environment, which you can do with the command:

source Artifact/bin/activate

Lastly, you can run the artifact program with the command: *python3 Artifact.py*

A.3.2 Basic Test

[Mandatory] There is no real way to run a basic test without running the entire program. But on the first time running the program, it may take up to one full minute before you start to see output. Once you start seeing lines appear stating *Building: [DATAFILE]*, then you know the artifact is running properly and able to read in the participant data properly.

A.4 Evaluation workflow

[Mandatory] After running the artifact, there are two locations with output that should be validated: the console where you ran the program, and a newly created output folder called "ArtifactOutput. The Code for generating the figure begins on line 1215."

A.4.1 Major Claims

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available]

(C1): *Figure 2 showing the number of participants with different counts of Mistakes and Web Pages by problem step. This output is reproduced in "CodeTypeByStep.png" in the ArtifactOutput folder.*

(C2): *Table 3 showing the counts of different events broken down by which step they occurred during. This output is given in the file CountByStep.csv generated in the ArtifactOutput folder, and the code for calculating it begins on line 1307.*

(C3): *Throughout the results, there are references to Chi-squared test results. The full table of those tests are shown in the supplemental PDF aptly named "supplemental.pdf" in the same artifact OSF folder. These numbers are displayed on the console and the code for calculating them begins on line 1086.*

In addition to the data related to the major claims, there is a folder generated in the "ArtifactOutput" folder, which contains additional descriptive figures about participant interviews that were used for axial coding but ultimately did not explicitly make it into the final paper. As an example, there is one graph displayed that visualizes the actions a participant took through the duration of a given step. For more information, check the README.txt file provided in the OSF repository.

A.4.2 Experiments

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available]

(E1): *Statistical results from analysis.* There are no specific experiments to run for our paper per say, but all of the major claims can be validated within the one provided script. It should take no more than 10 human-minutes to properly set up the environment, and five compute minutes to process the data. Please follow the instructions in this document and the README in the artifact folder for details about running the file.

A.5 Notes on Reusability

[Optional] Our artifact shows a potential first pass at our rich data set to extract insights. Others could reuse this artifact to investigate additional facets of beginner struggles. The data structures and data processing defined in the first 1000 lines is a starting point for investigating more detailed questions or replicating this study with our codebook and coding practices in a new environment.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: *"I have no idea how to make it safer"*: Studying Security and Privacy Mindsets of Browser Extension Developers

Shubham Agarwal, Rafael Mrowczynski, Maria Hellenthal, Ben Stock

CISPA Helmholtz Center for Information Security, Saarbruecken, Germany
{shubham.agarwal, mrowczynski, hellenthal, stock}@cispa.de

A Artifact Appendix

A.1 Abstract

We studied the security and privacy mindset of browser extension developers by conducting a semi-structured interview with them. As presented in the main paper, we interviewed 21 extension developers and observed that they only have surface knowledge of S&P issues and threats associated with extensions. To worsen the situation, they reported having little to no idea on how to best these threats in most cases.

We open-source all the artifacts associated with the design of the interviews – the interview guide, the code book, the email invites used for recruitment, and the pre-screening survey questionnaire used to select candidates. Notably, our semi-structured interview also required the participants to work on a set of coding tasks to assess their working knowledge of security and privacy practices. Hence, we also open source the prototypical Web application and the set of browser extensions used for individual coding tasks in their functional state for transparency and reuse in the future.

A.2 Description & Requirements

The functional components of our artifact include a prototypical e-commerce application and the set of browser extensions that we created to facilitate the coding tasks. The Web application is based on the Django framework and includes a `docker-compose.yml` file that allows quick and easy deployment on any host machine, with `docker` installed on it.

Further, each of the browser extensions contains logic to perform interactions with the DOM of the visited Web page (e.g., create buttons, inject iframes, etc.) based on the corresponding coding tasks. Thus, to verify their functionality, it is necessary to load individual extensions into the browser, and hence, a headful evaluation environment is required. However, the browser extensions are platform-agnostic and are tested to work on the following browsers: *Google Chrome*, *Brave*, *Arc*, *Opera*, *Microsoft Edge* and *Mozilla Firefox*.

A.2.1 Security, privacy, and ethical concerns

There are no security and privacy concerns associated with our artifacts. The prototype Web application and the browser extensions do not collect any data or interact with any of *our* remote servers whatsoever. However, we note that the browser extensions, corresponding to *Coding Task 2* and *Coding Task 3*, send network requests with session cookies, if available, to `amazon.de` and `connect.facebook.com`, respectively, necessary for the functionality of the coding tasks. Thus, we advise our users and evaluators to load these extensions in a *clean/incognito* environment to avoid any privacy issues.

A.2.2 How to access

The artifact can be accessed and downloaded from the following link: `https://doi.org/10.5281/zenodo.15550089`.

A.2.3 Hardware dependencies

There are no hardware dependencies for our artifact other than the fact that the functionality of the browser extensions can only be observed in a headful environment (i.e., PC or laptop with display and *not* a remote server).

A.2.4 Software dependencies

The operating system environment should allow running (docker) container environment. Alternatively, as we detail later, one can run the Web applications without docker by installing Django and other necessary Python packages directly on their host machine. Further, as mentioned before, the functional evaluation of browser extensions requires them to be loaded into the browser. Thus, at least one of the following browsers must be installed on the machine: *Google Chrome*, *Brave*, *Arc*, *Opera*, *Microsoft Edge* and *Mozilla Firefox*.

A.2.5 Benchmarks

None.

A.3 Set-up

We now outline the steps to set up the evaluation environment for the Web application and the browser extensions. As mentioned above, the tests only require the Web application to be running (either through the docker container or natively) and a browser that supports extensions. Once the artifact is downloaded, please unzip the following folders for evaluation: `dev_study.zip` and `extension_coding_tasks.zip`.

A.3.1 Installation

We assume our users/evaluators already have one of the required browsers installed on their machine. If not, we will allow them to install the browser they choose for the tests. Next, to enable loading unpublished browser extensions locally, please follow the following steps for Chromium browsers:

1. Navigate to `chrome://extensions`.
2. Toggle the **Developer Mode** button on the top-right corner of the page to switch it on. This allows browser extensions to be loaded from a local machine.

Firefox does not require any manual configurations to load extensions locally.

To install docker, please follow the instructions on the official documentation based on the host environment, accessible at `https://docs.docker.com/engine/install/`. After installing the docker, please ensure that the docker daemon is already running in the background, as follows:

```
$ docker --version
```

`[Docker Alternative]`: If the host environment does not allow running docker containers, one can run the Web application natively through Python. For this, they need to install the related project dependencies first, as follows:

```
$ pwd
./extension-developer-study

$ cd dev_study/apps/app1/
$ pip install -r requirements.txt
```

A.3.2 Basic Test

To run the Web application on your machine:

- First, navigate to the application directory
  `$ cd dev_study`
- Next, run the docker container.
  `$ docker compose up --build -d`
- Alternatively, if you plan to run the Web application natively, execute the following:
  `$ cd apps/app1/`
  `$ python manage.py runserver 9000`

Now, the Web applications should be running and accessible at: `http://127.0.0.1:9000`. Default login credentials: username: `foo`, password: `bar`.

Note: While running the application in a docker-less fashion, please ensure that the `python` version is configured correctly to `>=3.10` for compatibility with dependencies. The default port number specified within the `docker-compose.yml` files is 9000. So, please ensure the port is available for use or set the desired port number in the corresponding files.

Next, to load any of the browser extensions into the browser:

For Chromium browsers:

1. Navigate to `chrome://extensions`.
2. Then, click **Load unpacked** and select one of the extension directories (e.g., extension_coding_tasks/primary/task1).

For Mozilla Firefox:

1. Navigate to `about:debugging#/runtime/this-firefox`.
2. Then, click **Load Temporary Add-on...** and select the manifest of one of the extensions (e.g., extension_coding_tasks/primary/task2/manifest.json).

The extension should be loaded without errors/warnings. The functional test environment is now set up.

A.4 Evaluation workflow

A.4.1 Major Claims

The following are the major claims that need to be evaluated for the functionality of our artifact:

(C1): The prototypical Web application is functional and contains necessary pages required for the coding tasks.

(C2): The browser extensions corresponding to each coding task is functional and injects necessary DOM elements into the e-commerce application above.

Note: The browser extensions are only functional to the point that it injects DOM elements into the appropriate page (in E1), and includes partial logic to perform operations (in E2 & E3). The study participants implemented the rest of the S&P-related logic, as listed in Table 1 of the main paper.

A.4.2 Experiments

(E1): *[Coding Task 1] [<5 human-minutes]:*

- Load the `task1` extension from `extension_coding_task/primary/task1/` into the browser.
- After running the Web application locally, navigate to `http://127.0.0.1:9000` and login with default credentials.
- Navigate to `http://127.0.0.1:9000/products` or click on the *Product* tab on top. Then, click on the *Order Now* button for any of the listed items to navigate to the *Order* page.
- The `task1` extension should inject two additional buttons – *Save Address for Later* and *Autofill Address Data*, before the *Continue to Payment* button (as shown in Figure 1). Disabling the extension will remove the two injected buttons from the UI.

(E2): *[Coding Task 2] [<5 human-minutes]:*

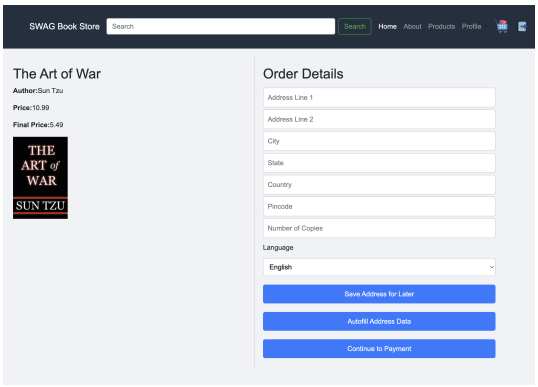


Figure 1: Expected view of the *order* page when `task1` extension is loaded.

- Load the `task2` extension from `extension_coding_task/primary/task2/` into the browser.
- Navigate to `http://127.0.0.1:9000/products` or click on the *Product* tab.
- The browser extension injects an additional button – `Check on Amazon` below the `Order Now` for each listed product, as shown in Figure 2. This button and the associated logic are supposed to allow checking the price of the listed product on *Amazon*.
- However, when clicking on the injected button, the *iframe* does not work since the interview participants are supposed to fix the issue. Disabling the corresponding extension will remove the button from the UI.

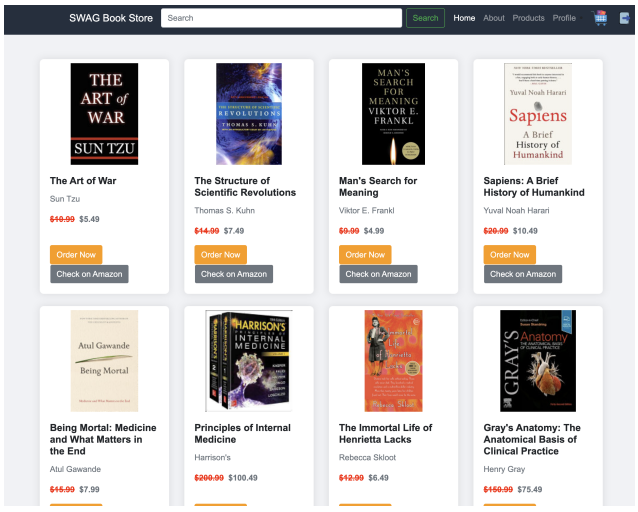


Figure 2: Expected view of the *product* page when `task2` extension is loaded.

(E3): [Optional]: *[Coding Task 3] [<5 human-minutes]:*

- Load the `task3` extension from `extension_coding_task/primary/task3/` into the browser.
- Navigate to `http://127.0.0.1:9000/products` or click on the *Product* tab.
- Here, since the application sends the CSP header, which blocks all third-party scripts from being executed. Thus, the page UI has no visual cues to verify the extension's functionality.
- However, one can open the *Developer Tools* and verify the functionality by looking for the corresponding error in the console – *inject.js:10 Refused to load the script 'https://connect.facebook.net/en_US/sdk.js' because it violates the following Content Security Policy directive: "script-src 'self'". Note that 'script-src-elem' was not explicitly set, so 'script-src' is used as a fallback.*
- Alternatively, one can also inspect the outgoing network logs to `https://connect.facebook.net/en_US/sdk.js`.

A.5 Notes on Reusability

The Web Application & Browser Extensions: The prototypical e-commerce application could very well be extended with more functionalities and pages without actually capturing any data based on the requirements and research focus. Similarly, the open-sourced extension boilerplate could be used as is or modified accordingly to interact with the e-commerce application developed here. However, we strictly advise against using our extension source code for real-world extension development or publishing, as the manifest of these extensions includes redundant host and API permissions.

Interview Guide: Our interview guide elaborates on the structure, topical areas, and related questions we asked our participants in our semi-structured interviews. Future studies may build upon this guide for similar studies on browser extensions, or, in general that may include coding tasks.

Email Invites: The content of the email invites that we sent to our participants only partially disclosed our background. That is, we only identified ourselves as *Computer Scientists* and not *Security/Privacy experts* to eliminate any potential bias during recruitment or in later phases of our study, however, without compromising the ethical boundaries of recruitment. Researchers may reuse our email content after appropriate changes to recruit developers under similar constraints.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2025/`.





USENIX Security '25 Artifact Appendix: Precise and Effective Gadget Chain Mining through Deserialization Guided Call Graph Construction

Yiheng Zhang[1,2], Ming Wen[1,2,*], Shunjie Liu[2], Dongjie He[4], and Hai Jin[1,3]

[1]*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST)*
[2]*Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, HUST, China*
[3]*Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, China*
[4]*Chongqing University, China*

A Artifact Appendix

A.1 Abstract

Our artifact, FLASH, is a tool designed to detect potential gadget chains in Java applications by constructing deserialization-guided call graphs. FLASH begins with a novel Deserialization-Driven Controllability Analysis, which determines whether variables can be restored through deserialization. Based on the controllability of variables, FLASH performs a series of downstream tasks, including hybrid dispatch and reflection analysis, thereby building a more comprehensive and precise deserialization call graph. Utilizing the deserialization call graph, FLASH applies an existing detection algorithm [1] to identify gadget chains.

Evaluation conducted on benchmarks demonstrates that FLASH achieves higher recall (with 30.8% lower false negative rate) and precision (with 25.9% lower false positive rate) than state-of-the-art methods in gadget chain detection, at the cost of limited overhead. Moreover, FLASH successfully discovers 90 new gadget chains.

A.2 Description & Requirements

This section provides essential information regarding the usage of FLASH, including its access link, required software dependencies such as the Java runtime environment or Docker, the location of the benchmark used in our experiments, and safety considerations when verifying the exploitability of detected gadget chains.

A.2.1 Security, Privacy, and Ethical concerns

It is important to note that the execution of a gadget chain ultimately triggers dangerous functions. Therefore, when verifying the exploitability of a gadget chain, we replace the malicious payload with a benign command (e.g., opening a calculator or referencing a non-existent address) to ensure safety.

*Ming Wen is the corresponding author

A.2.2 How to Access

We have open-sourced FLASH on Zenodo at `https://doi.org/10.5281/zenodo.15606159`, where it is publicly available for download.

A.2.3 Hardware Dependencies

None

A.2.4 Software Dependencies

If users choose to compile and run FLASH from source, a JDK 17 environment is required. Alternatively, we provide a pre-built executable version of FLASH and a Docker-based environment, allowing users to run the tool with only Docker installed. Additionally, to detect gadget chains under a specific JDK version, users need to require the corresponding JDK dependency libraries, which are essential for FLASH to detect gadget chains and validate the exploitability of the identified gadget chains.

A.2.5 Benchmarks

To facilitate evaluation, we have open-sourced our benchmark of Java applications containing our detected new gadget chains in the `Flash_GC.zip` file. FLASH allows users to specify target Java applications for analysis and automatically detects gadget chains contained in these applications.

A.3 Set-up

This subsection primarily introduces the set-up steps required before using FLASH.

A.3.1 Installation

We recommend using IntelliJ IDEA and JDK 17 to run FLASH from source code. Specifically, after downloading the source code, users can open the project directly in IntelliJ IDEA. Then, users can follow the configuration steps for the Tai-e framework [3] to set up the project, as described in the official documentation [2].

A.3.2 Basic Test

To verify that all required components for FLASH are properly set up, users can perform a functionality test by analyzing a sample Java application, which requires specifying runtime parameters in a YAML configuration file. We list and explain some of the key configuration parameters as follows:

- `appClassPath`. Users need to specify the target Java application to be analyzed.
- `sources`. Users need to specify the entry points for analysis. The keyword `serializable` can be used to include all deserialization-related methods, such as `readObject`, `readExternal`. Alternatively, users can specify other methods for analysis by providing their method signatures.
- `filterNonSerializable`. Users can configure to filter out methods whose declaring classes do not implement `java.io.Serializable`.
- `GC_MAX_LEN`. Users can configure the maximum length of a gadget chain that FLASH will search for during analysis.
- `priori-knowledge`. Users need to specify the file that contains manually created summaries for JDK built-in APIs (e.g., String and JavaBean).

We include a sample configuration file in our artifact to help users get started. Users can configure IntelliJ IDEA to pass this file to FLASH by specifying it with the `-options-file` parameter. We also recommend setting VM options such as `-Xss` or `-Xmx` to allocate additional memory to FLASH for better performance. If FLASH outputs runtime logs (e.g., the loading of sinks and the number of analyzed methods), it indicates that it functions correctly and is ready for use.

A.4 Evaluation Workflow

Since our primary focus is on gadget chain detection, this subsection mainly evaluates the functionality of FLASH in detecting gadget chains.

A.4.1 Major Claims

(C1): Compared to the state-of-the-art approaches, FLASH achieves higher recall (with 30.8% lower false negative rate) and precision (with 25.9% lower false positive rate) in gadget chain detection. In total, it identifies 90 new gadget chains. This is demonstrated by experiment (E1), as described in Section A.4.2. The results are also presented in Table 1 of our paper.

A.4.2 Experiments

(E1): [Gadget Chain Detection] [30 human-minutes + 1 compute-hour + 1GB disk]: This experiment focuses on detecting potential gadget chains within a specified application. The expected results are the concrete gadget chain call stacks. Note that the time estimation refers to the average time required for analyzing a single target.
Preparation: Users should configure the required parameters (e.g., the target application to be analyzed) in accordance with the procedure outlined in Section A.3.2.
Execution: FLASH can be launched by simply running the IntelliJ IDEA.
Results: The expected results of the experiment are concrete gadget chains, represented as method call sequences. For instance: `Source#readObject` → `Link1#callee` → `Sink#sink`. It is worth noting that manual validation is necessary for the detection results. To facilitate this, users may refer to the gadget chains and corresponding proof-of-concepts (POCs) examples included in our benchmark as a basis for constructing and testing new exploit POCs.

To simplify large-scale analysis, we provide a pre-configured Docker environment and an accompanying bash script that automates the gadget chain detection of all Java applications within a specified directory. If you run the script, it will naturally require more time.

A.5 Notes on Reusability

First, the deserialization call graph generated by FLASH can serve as a foundational component for other research directions, such as fuzzing-based validation. Users can also easily export and store the generated call graph for further analysis.

Second, FLASH can be easily extended to support other types of vulnerability detection, such as web injection attacks. By modifying the configuration file to specify a web API as the source, and disabling the deserialization-specific analysis in the code, users can adapt FLASH to new detection scenarios with minimal efforts.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodol-

ogy followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] Xingchen Chen, Baizhu Wang, Ze Jin, Yun Feng, Xianglong Li, Xincheng Feng, and Qixu Liu. Tabby: Automated gadget chain detection for java deserialization vulnerabilities. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 179–192. IEEE, 2023.

[2] Tian Tan. Tai-e Reference Documentation, 2025. https://tai-e.pascal-lab.net/docs/current/reference/en/index-single.html#setup-tai-e-in-intellij-idea.

[3] Tian Tan and Yue Li. Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1093–1105, 2023.



USENIX Security '25 Artifact Appendix: Aion: Robust and Efficient Multi-Round Single-Mask Secure Aggregation Against Malicious Participants

Yizhong Liu[†], Zixiao Jia[†], Zian Jin[†], Xiao Chen[†], Song Bian[†], Runhua Xu[ᛤ], Dawei Li[†*], Jianwei Liu[†*], Yuan Lu[ᛋ]

[†]School of Cyber Science and Technology, Beihang University, [ᛤ]School of Computer Science and Technology, Beihang University, [ᛋ]Institute of Software, Chinese Academy of Sciences

Email: {liuyizhong, jiazixiao, jinzian, chenxiao, sbian, runhua, lidawei, liujianwei}@buaa.edu.cn, luyuan@iscas.ac.cn

A Artifact Appendix

A.1 Abstract

This paper introduces Aion, a multi-round single-mask secure aggregation scheme with evolving input validation against malicious clients and aggregators. The artifact includes implementations of the core components evaluated in our paper, covering the following experiments: Secure Aggregation Protocol Simulation **(E1)**, Input Validation Evaluation **(E2)**, and Gradient Inversion Attack **(E3)**. In **E1**, we evaluate Aion's performance and robustness in terms of execution time, message overhead, and network resilience under varying system parameters. **E2** evaluates Aion's L_2-norm-based input validation against poisoning attacks on multiple datasets. **E3** reproduces a gradient inversion attack with access to labels and BatchNorm statistics, demonstrating the effectiveness of Aion's mask-based privacy protection.

A.2 Description & Requirements

This section details the experimental setup, specifying the hardware and software environments, as well as the benchmarks employed to generate the reported results.

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns associated with the execution of this artifact.

A.2.2 How to access

Our artifact is available at https://doi.org/10.5281/zenodo.15605465.

A.2.3 Hardware dependencies

Evaluating our artifact requires a system equipped with at least an NVIDIA GeForce RTX 3060 GPU and with no specific CPU requirements. However, for efficiency, we recommend using a higher configuration, e.g., NVIDIA GeForce RTX 4090 GPU and Intel Core i9-14900KF.

A.2.4 Software dependencies

E1 is compatible with both Windows and Linux, except for the ACORN scheme, which is only supported on Linux. **E2** is compatible with Windows, while **E3** is for Linux. We strongly recommend deploying **E2** and **E3** using Docker, which is compatible with both Windows and Linux. All required dependencies are listed in the `requirements.txt`.

A.2.5 Benchmarks

Our experiments require the CIFAR10, FMNIST, EMNIST-Byclass, and SHAKESPEARE datasets, which are automatically downloaded from their official sources during code execution. The ResNet-18, LeNet-5, and ResNet-9 models are used in **E2** and **E3**.

A.3 Set-up

This section provides the steps necessary to install and configure the environment for evaluating the artifact.

A.3.1 Installation

We use Anaconda to set up the environment of **E1**.

1. Create a new Conda environment named aion and activate it: `conda create --name aion python=3.8`
`conda activate aion`
2. Use pip to install the required packages: `pip install -r requirements.txt`

We recommend using Docker for installation of **E2** and **E3**. The pre-built images can be downloaded as follows:

E2: `docker pull aionaion/input_validation:latest`
E3: `docker pull aionaion/gradattack:latest`

A.3.2 Basic Test

For **E1**, enter the folder `pki_files` and run `setup_pki.py`:
`cd pki_files`

```
python setup_pki.py
cd ..
```

E1 has multiple configs.

```
-c [protocol name]
-n [number of clients (power of 2)]
-i [number of iterations (training rounds)]
-A [number of aggregators]
```

Aion supports batches of clients with size power of 2, e.g., 128, 256, 512, 1024, 2048, 4096.

Example command:

```
python abides.py -c aion -n 128 -A 8 -i 10
```

For **E2** and **E3**, once the image is pulled, activate the main container to run the functionality test:

E2: `docker run --gpus all --rm aionaion/input_validation:latest`

E3: `docker run --gpus all --rm aionaion/gradattack:latest`

You can also verify that all required dependencies are correctly installed without using Docker:

E2: `python ./FL_Backdoor_CV/roles/autorun.py`

E3: `python examples/attack_cifar10_gradinversion.py #remove spaces in file path`

For **E2**, if the training progresses to Round 301, you can safely terminate the process. For **E3**, we recommend waiting until the attack completes to inspect the reconstructed image.

A.4 Evaluation workflow

In this section, we present the main claims of the paper along with the experiments designed to support them.

A.4.1 Major Claims

The major claims presented in our paper are as follows:

(C1): Aion consistently demonstrates superior performance in terms of execution time and communication overhead compared to state-of-the-art schemes across varying numbers of clients (q) and aggregators (n). This is proven by the experiment (**E1.1**) described in Section 7.1, whose results are illustrated in Figure 2 (for varying q) and Figure 3 (for varying n).

(C2): Aion maintains robust and efficient performance even as the security parameter for secret sharing (p) increases, showing significantly less performance degradation compared to competing schemes. This is proven by the experiment (**E1.2**) described in the supplementary material (Appendix C), whose results are illustrated in Figure 13.

(C3): Aion exhibits strong resilience and superior time efficiency under adverse network conditions, consistently outperforming other masking-based schemes. This is proven by the experiment (**E1.3**) described in the supplementary material (Appendix C), whose results are illustrated in Figure 14.

(C4): Aion effectively defends against poisoning attacks when the poisoning ratio is below 50%. This is proved by experiment (**E2**), described in Section 7.2, with the corresponding results presented in Figures 4 and 5.

(C5): Aion is shown to be robust against gradient inversion attacks even when attackers have access to private image labels and BatchNorm statistics, as evidenced by experiment (**E3**) in Appendix F.2 and the corresponding results in Figure 16.

A.4.2 Experiments

The following experiments provide supporting evidence for the main claims of the paper.

(E1.1): This experiment aims to verify that Aion has superior performance in terms of execution time and message overhead compared to state-of-the-art schemes (Flamingo, ACORN, SecAgg+) when varying the number of clients (q) and aggregators (n).

Execution: To reproduce the results for varying client numbers (Figure 2), run the following commands, iterating through the number of clients `-n` with values of 128, 256, 512, 1024. The number of aggregators is fixed at 8.

```
python abides.py -c aion -n 128 -A 8 -i 10
python abides.py -c aion -n 256 -A 8 -i 10
python abides.py -c aion -n 512 -A 8 -i 10
python abides.py -c aion -n 1024 -A 8 -i 10
```

To reproduce the results for varying aggregator numbers (Figure 3), run the following commands, iterating through the number of aggregators `-A` with values of 8, 16, 32, 64. The number of clients `-n` is fixed at 512.

```
python abides.py -c aion -n 512 -A 8 -i 10
python abides.py -c aion -n 512 -A 16 -i 10
python abides.py -c aion -n 512 -A 32 -i 10
python abides.py -c aion -n 512 -A 64 -i 10
```

Results: The script will output the execution time (in seconds) and message overhead (in bytes) for the initialization and aggregation phases. You can plot these values to reproduce Figures 2 and 3. The performance is expected to surpass the results of the baseline schemes reported in the figures.

(E1.2): This experiment verifies that even as the security parameter p (the prime size for secret sharing) increases, Aion's performance remains robust and efficient, degrading much more gracefully than competing schemes.

Execution: Vary the `--prime_bits` with values of 2048, 3072, 4096, with the number of clients `-n` set to 256 and the number of aggregators `-A` set to 8.

```
python abides.py -c aion -n 256 -A 8 -i 10
--prime_bits 2048
python abides.py -c aion -n 256 -A 8 -i 10
--prime_bits 3072
python abides.py -c aion -n 256 -A 8 -i 10
--prime_bits 4096
```

Results: The script will report the initialization and aggregation times for each configuration. After running all commands, plot the execution times against the prime

sizes to reproduce Figure 13. The performance is expected to surpass the results of the baseline schemes reported in the figure.

(E1.3): This experiment shows Aion's strong resilience and superior time efficiency under adverse network conditions, including high latency and low bandwidth.

Execution: Varying latency (corresponding to Figures 14a, 14b):

For each scheme, run the script using different `--latency` values (e.g., 10, 100 ms):

```
python abides.py -c aion -n 512 -A 8 -i 10
--latency 10
python abides.py -c aion -n 512 -A 8 -i 10
--latency 100
```

Varying bandwidth (corresponding to Figures 14c, 14d):

For each scheme, run the script using different `--bandwidth` values (e.g., 1, 5, 10 Mbps):

```
python abides.py -c aion -n 512 -A 8 -i 10
--bandwidth 1
python abides.py -c aion -n 512 -A 8 -i 10
--bandwidth 5
python abides.py -c aion -n 512 -A 8 -i 10
--bandwidth 10
```

Results: The script will output the total time overhead (computation + communication) for the initialization and aggregation phases under the simulated network conditions. By plotting these results, you can reproduce Figure 14. Aion is expected to have the lowest time overhead across all tested conditions, with its advantage being most pronounced in challenging environments such as high latency and low bandwidth.

(E2): This experiment evaluates whether Aion can effectively defend against poisoning attacks.

How to: To run the experiments, Docker users should enter the container with: `docker run --gpus all -it aionaion/input_validation:latest /bin/bash`. Then execute the scripts in the `roles` folder (e.g., `attack1_cifar10.py`). You can adjust the poisoning ratio using `number_of_adversaries`, the mask ratio using `weight`, and the boost rate of malicious gradients using `mal_boost`.

Execution: Please run the command:

```
python ./FL_Backdoor_CV/roles/
attack1_cifar10.py  --aggregation_rule aion
--number_of_adversaries 5
#remove spaces in file path
```

This process is expected to take approximately 20 minutes. Upon completion, the ASR and TER values will be recorded in `.pt` files within the `results` folder. To reproduce the results shown in Figure 4, you can vary the `number_of_adversaries` parameter from 5 to 50.

```
python ./FL_Backdoor_CV/roles/
attack2_cifar10.py
--aggregation_rule aion --weight 0.1
python ./FL_Backdoor_CV/roles/
attack3_cifar10.py
--aggregation_rule aion --mal_boost 50
#remove spaces in file path
```

You can vary the `weight` parameter from 0.05 to 0.5 and the `mal_boost` parameter from 25 to 250. The results contribute to Aion curves presented in Figures 6 and 8.

Results: You can check the results by running: `python check_results.py`.

(E3): : This experiment aims to demonstrate that Aion is robust against gradient inversion attacks.

How to: To run the experiments, please enter the container's interactive environment. The following commands also mount a local `results` directory into the container to save attack outputs.

```
docker run --rm -it --gpus all -v "%cd%
results:/app/results" aionaion/gradattack
/bin/bash (For Windows)
sudo docker run --rm -it --gpus all
-v "$PWD/results":/app/results
aionaion/gradattack /bin/bash (For Linux)
```

To run the experiments, execute `attack_cifar10_gradinversion.py` in the `example` folder. Each run will generate a reconstructed image.

Preparation: You can optionally train the checkpoint using Aion as the target model.

```
python3 examples/train_cifar10.py
--n_epoch 48 --logname CIFAR10/Aion
--defense_aion --weight 0.1
```

However, this step is not required, as we have provided pre-trained checkpoints in the `checkpoint_old` folder.

Execution: Please run the command:

```
python3 examples/attack_cifar10_
gradinversion.py --batch_size 16 --BN_exact
--tv 0.1 --bn_reg 0.005 --defense_aion
--weight 0.1 --attack_checkpoint
checkpoint_old/aion_epoch=48.ckpt
#remove spaces in file path
```

To perform an attack on a newly trained model, be sure to update the `attack_checkpoint` parameter accordingly. Note that newly generated checkpoints are saved in the `checkpoint` folder.

Results: This process is expected to take approximately 10 minutes. Upon completion, the reconstructed image, `reconstructed.png`, will be saved in the `results` folder on your local machine (the one you mapped to the Docker container).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Doubly Dangerous: Evading Phishing Reporting Systems by Leveraging Email Tracking Techniques

Anish Chand
Louisiana State University

Nick Nikiforakis
Stony Brook University

Phani Vadrevu
Louisiana State University

A Artifact Appendix

A.1 Abstract

In our paper, we repurposed various email tracking vectors to profile and evade the anti-phishing systems employed by major email service providers. We identified the different HTTP headers and tracking vectors triggered by the email services during different stages like email prefetching, email opening and after email reporting. Due to the distinct behavior exhibited by the email systems, we were able to identify and evade them using evasive phishing attacks. The artifact provided contains the dataset and the code used for the project. The dataset is accompanied by a helper script to execute queries to generate the results presented in the paper. README files are also included to facilitate easy navigation and reproduction of results found in the paper.

A.2 Description & Requirements

To support these findings, this artifact is provided in two main components:

1. (Dataset and Results): We provide the dataset in a self-contained Docker environment that includes a PostgreSQL database injected with the dataset from our experiments. We also provide an accompanying Python script that allows for the reproducibility of the major tables and figures presented in our paper.

2. Part B (Code): The code provided includes three software components: email-sending module, a websites module and a GUI automation module. This part uses Docker to run the database but runs our three main applications (two Laravel web apps, one Python automation script) locally on host machine. These applications run locally with a small, artificial database to demonstrate their architecture and testability without requiring live credentials or external network access such as live web servers and domains.

A.2.1 Security, privacy, and ethical concerns

Throughout our research, we took several measures to mitigate any security, privacy or ethical concerns. Our investigation findings pose security risks to the email providers if they are exploited and is not intended for unauthorized or malicious actors. Hence, we have made the dataset and the websites code module that exploits the dataset to launch evasive phishing campaigns are available with restricted access on Zenodo. Additionally, the provided applications are operating locally without the ability to perform live actions such as sending emails and launching phishing sites. There are no destructive steps, and no security mechanisms on the evaluator's machine need to be disabled.

A.2.2 How to access

Our artifact is available on Zenodo. Our public version with the email sending code and the GUI automation code are available here: https://doi.org/10.5281/zenodo.15612284. Our restricted repository that contains the dataset, websites module code, email sending code and GUI automation code are available here: https://doi.org/10.5281/zenodo.15612193

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

The following tools are required to be installed on your device:

- **Operating System:** Linux (tested on Ubuntu 22.04) or macOS (tested on macOS 15.5). The GUI automation script has only been tested on Windows 11.
- **Docker and Docker Compose:** For running the PostgreSQL database containers.
- **PHP 8.3+** and **Composer 2.x**: For running the Laravel applications locally in Part B.
- **Python 3.9+** and **virtualenv**: For running the analysis and automation scripts.
- **The 'whois' command-line utility:** This is a mandatory dependency for Part B (the Functional Demo). Our Laravel application uses it to perform lookups on IP addresses.

A.2.5 Benchmarks

None.

A.3 Set-up

This section provides the complete installation and configuration steps for the artifact.

A.3.1 Installation

This setup allows for the direct regeneration of our paper's results from the provided dataset.

1. **Start the Environment:** Navigate to the `results` directory. Then, execute the following command to start the PostgreSQL container and automatically load the full dataset:

   ```
   docker-compose up --build -d
   ```

2. **Install Python Dependencies:** Navigate to the `results/analysis` directory and run:

   ```
   pip install -r requirements.txt
   ```

3. **Start the Database:** Navigate to the `code` directory and start the PostgreSQL container:

   ```
   docker-compose up -d
   ```

4. **Set up the email-sender and database:** Go to the application directory `email-sending`, install dependencies and configure the application:

   ```
   composer install
   cp .env.example .env
   php artisan key:generate
   php artisan migrate --seed
   ```

   Expected Output: The console will show successful migration and seeding messages. The database is now correctly structured and populated for the demo.

5. **Run the websites Application:** This application serves the web pages used for phishing simulation and honey sites. Navigate to its directory, `websites-code`, install its dependencies and then start the local web server:

   ```
   composer install
   cp .env.example .env
   php artisan key:generate
   php artisan serve --port=8000
   ```

A.3.2 Basic Test

1. **Test for dataset evaluation:** To verify that the analysis script is functional in *results/analysis*, run its help command:

   ```
   python queries.py -h
   ```

   Expected Output: The script will print help and options to the console.

2. **Test for websites-code:** Open a web browser and navigate to `http://localhost:8000` after `php artisan serve -port=8000`.

   Expected Output: The application's homepage should load successfully in the browser, confirming that the web server is running and connected to the demo database.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The major email providers namely Gmail, Outlook, and Proton, all employ email subsystems that can be identified using a combination of unique HTTP headers and tracking vectors. Tables 7, 8, and 9 showing unique HTTP headers and Tables 4, 5, and 6 showing the unique tracking vectors triggered highlight this.*

(C2): *The tracking vectors and HTTP headers can be used to launch evasive phishing sites that can avoid detection from the email anti-phishing entities compared to the baseline normal sites.*

A.4.2 Experiments

(E1): *There are no experiments that can be run. However, all of our major claims can be validated with the provided queries.py script. The instructions to setup are in the README file in the folder and the environment can be set up in a matter of some minutes. The script will output the tables in the console and may output figure when applicable.*

(E2): *Similar to above, this claim can be validated with the provided queries.py script that will output the number of email reports made for the smart site, which is significantly higher compared to the baseline sites. The provided websites-code can be used to evaluate the evasion logic used compared to the baseline sites as well as the phishing payload used. The accompanying README in the folder provides additional steps that may be helpful in evaluation of the code.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodol-

ogy followed for the evaluation of this artifact can be found at
https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Evaluating the Effectiveness and Robustness of Visual Similarity-based Phishing Detection Models

Fujiao Ji[1], Kiho Lee[1], Hyungjoon Koo[2], Wenhao You[3], Euijin Choo[3], Hyoungshick Kim[2], Doowon Kim[1]

[1]*University of Tennessee, Knoxville* [2]*Sungkyunkwan University* [3]*University of Alberta*

A Artifact Appendix

A.1 Abstract

Phishing attacks pose a significant threat to Internet users, with cybercriminals elaborately replicating the visual appearance of legitimate websites to deceive victims. Visual similarity-based detection systems have emerged as an effective countermeasure, but their effectiveness and robustness in real-world scenarios have been underexplored. In this paper, we comprehensively scrutinize and evaluate the effectiveness and robustness of popular visual similarity-based anti-phishing models. To facilitate reproducibility and accelerate scientific progress–particularly in strengthening collective efforts in combating phishing attacks–we share the resources while adhering to ethical considerations. Given that our work primarily involves evaluation, encompassing multiple datasets and experiments, this artifact appendix highlights key claims and experiments conducted with the Phishpedia detection model to demonstrate the process. Detailed information is available in our GitHub repository https://github.com/Fujiaoji/PhishingEval.

A.2 Description & Requirements

In this section, we provide the essential details needed to replicate the same experimental setup and execute the artifact.

A.2.1 Security, privacy, and ethical concerns

Our research involves the APWG eCX phishing dataset. To maintain ethical standards, we provide HTML, screenshots, and domains without revealing URLs. Furthermore, we utilize open-source models and websites.

Note that, the apwg451514 dataset requires a large amount of space. Additionally, when running the 'DynaPhish' model, please avoid using large datasets, as exceeding the free account credits will result in additional charges.

A.2.2 How to access

We share the resources on our website (https://moa-lab.net/evaluation-visual-similarity-based-phishing-detection-models/) and Zenodo (https://doi.org/10.5281/zenodo.14804193). The code is available on our GitHub repository (https://github.com/Fujiaoji/PhishingEval). Note that, we share the apwg451514 dataset through OneDrive since it occupies a very large space. Please fill out the form on our website to access this share.

A.2.3 Hardware dependencies

We run the codes on both CPU and GPU (NVIDIA A30).

A.2.4 Software dependencies

We run the codes on 'Ubuntu 24.04.1 LTS' and organize the environments through Anaconda. DynaPhish requires the online search and therefore needs to register the Google Cloud account and a browser for execution. Since there are seven baseline models and each model requires a distinct runtime environment, we therefore provide detailed installation instructions a bash file and respective txt file. Due to the page limitations, we share more details in the 'README.md' file on GitHub.

A.2.5 Benchmarks

The artifact mainly contains three components: **(I)** our collected datasets, **(II)** codes, and **(III)** retrained models.

Specifically, the **(I)** collected datasets consist of eight sections: (1) the APWG dataset (apwg451514, Table 2), (2) a sampled subset of the APWG dataset (phishing4190, Table 3), (3) a selected failed 6000 examples from APWG dataset (failed_examples_csv, Table 3), (4) a general benign dataset covering 100 domains (archive100, Table 4), (5) a benign dataset for 110 common brands between phishing datasets and reference lists, also used for ablation studies and manipulations (crawl_benign, Table 7, Table 8, and Table 9), (6) reference lists (expand277, expand277_new, merge277,

merge277_new), (7) a visible manipulation dataset (visible_dataset2, Table 8 and Table 9), and (8) a perturbed dataset (perturbated_dataset, Table 8 and Table 9).

The **(II)** code repository (PhishingEval) contains (1) seven codes of detection models used for evaluating different datasets, (2) crawler.py, a script used for data collection, (3) run_fastdup_cluster.py used for clustering and helping us filter the dataset, and (4) data_test provides three samples for testing.

The **(III)** retrained models are shared in the Zenodo. You can download it for each through the 'download_model.sh' file. Or you can download the folder manually and put them under appropriate positions based on the model structures of the 'README.md' file in the GitHub.

A.3 Set-up

In this section, we outline the installation and configuration steps for Phishpedia, as the setup steps for seven models follow a similar process.

A.3.1 Installation

First, download the repository as a '.zip' file instead of using 'git clone', as the dataset was previously uploaded to GitHub, resulting in extensive history. Follow the total structure outlined in the 'README.md' file to properly download and organize the folders. To run a small sample in the data_test, you need to download at least the corresponding target list datasets through the following command `bash download_data.sh <dataset name>`. For example, Phishpedia requires the logo-based target lists dataset, which can be downloaded through `bash download_data.sh expand277_new`.

After obtaining the code and datasets, the environment can be set up following these steps. Some packages are extracted to mitigate environmental conflicts, making them easier to use and replace.

```
cd PhishingEval/code/reproduce_phishpedia
bash download_model.sh # download models
bash setup_cpu.sh
conda activate env_phishpedia
```

A.3.2 Basic Test

The testing sample information is stored in the 'data_test.csv' file. To use different inputs, please set the parameter of 'input_csv' to your desired CSV path. The new CSV should follow the same column structure as the 'data_test.csv'. Then we can use the following command to run the example. Successful results will be saved in a CSV file. We provide the running code for testing sample and the phishing 4190 dataset in GitHub.

```
conda activate env_phishpedia
python eval_phishpedia.py -
    siamese_weights=models/bit.pth.tar -
    targetlist=../../data/targetlist/
    expand277 -input_csv=../../data/
    data_test/data_test.csv -input_folder
    =../../data/data_test
```

A.4 Evaluation workflow

Our work primarily focuses on evaluation. Reproducing the results requires running the entire set of experiments, which is time-consuming. To facilitate verification, we recommend using the sampled 'phishing4190', as all experiments follow a similar workflow. In this section, we provide some main claims and experiments. Additionally, we demonstrate the time and resource requirements based on the Phishpedia model.

A.4.1 Major Claims

(C1): Reference list-based models can introduce weaknesses. Logos or screenshots not included in the reference list but known to users may mislead the detection models. This highlights the need to constantly expand and update the reference lists and detection models. This is supported by the results in Section 5.1 (E1, E3), as shown in Table 2 and 3.

(C2): Phishpedia demonstrates a better detection rate on the D_{sample} compared to DynaPhish and PhishIntention. Inaccurate keyword extraction can degrade performance, while diverse screenshots and similar web designs present challenges to screenshot-based methods. Selecting an appropriate model structure is crucial to optimizing performance and mitigating these weaknesses. This is also proved by the results in Section 5.1 (E1, E3), as shown in Table 2 and 3.

(C3): Logo-based models currently offer the most reliable approach for standard phishing detection and brand identification, but they are susceptible to additional checking steps, used features, and logo components. Screenshot-based models struggle with web design diversity but may serve as a complementary solution for scenarios involving unknown or emerging brands. This is proved by the brand identification experiments (E2, E3) of Table 2 and 3 in Section 5.3.

(C4): Simple visible and perturbation-based manipulations significantly disrupt logo-based methods. Both are transferable. Screenshot-based methods maintain stable detection but struggle with identifying brands when logos are altered. This is proved by the robustness experiments (E4) of Table 8 and 9 in Section 6.1.

A.4.2 Experiments

After running the code, the results should be saved in a CSV file containing detection and/or image similarity outcomes. Based on these results, you can analyze the detection or identification performance. In this section, we highlight key experiments to provide further insights. The execution and results presented are based on Phishpedia with apwg451514 dataset. When running the code, you can replace the dataset with other smaller-size datasets.

(E1): *[Detection Effectiveness in Section 5.1] [30 human-minute + 42 compute-hour + 250GB disk]:*
Preparation: Download the apwg451514 dataset from Zenodo, prepare the input CSV files, prepare the models based on the set-up section, and ensure the availability of sufficient GPU, CPU, and disk resources.
Execution: Run the evaluation codes.
Results: After getting the detection results, apply predefined thresholds to decide the results. In this step, we assume that images with high similarity to the target will be thought of as phishing, as the entire dataset is phishing.

(E2): *[Phishing Brand Identification in Section 5.3] [60 human-minute + 42 compute-hour + 250GB disk]:* The process is similar to E1. However, after getting the results from models, it is necessary to determine if the predicted brands match the true target brands. Since the same brand may appear under different names, manual verification is required to ensure accuracy.

(E3): *[Detection and Identification Results on D_{sample} in Section 5.] [30 human-minute + 40 compute-minute + 30GB disk]:* Follows the same process as E1 and E2 but with a smaller sample dataset.

(E4): *[Robustness Evaluation in Section 6.1] [120 human-minute + 1 compute-hour + 45GB disk]:* The process contains both detection and identification on the manipulated dataset. Manual verification is also required to ensure the accuracy.

A.5 Notes on Reusability

It is important to make the name of brands consistent when dealing with datasets. Additionally, as original codes for cited papers are frequently updated, it is important to check for the latest versions to maintain compatibility and accuracy.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Hercules Droidot and the murder on the JNI Express

Luca Di Bartolomeo* EPFL
Philipp Mao* EPFL
Yu-Jye Tung UC Irvine
Jessy Ayala UC Irvine
Samuele Doria University of Padua
Paolo Celada EPFL
Marcel Busch EPFL
Joshua Garcia UC Irvine
Eleonora Losiouk University of Padua
Mathias Payer EPFL

A Artifact Appendix

A.1 Abstract

POIROT's artifact contains the source code necessary to run our Android native library fuzzer and static analysis passes. This document describes how to set up our prototype, and gives a brief overview of the resource requirements to replicate some of the experiments conducted in our evaluation, along with instructions to run them.

A.2 Description & Requirements

The artifact contains POIROT's source code and scripts to reproduce the evaluation.

A.2.1 Security, privacy, and ethical concerns

This artifact does not contain any threat to the system's integrity or privacy. However, we still recommend running it inside a sandboxed environment (either a VM or a container).

A.2.2 How to access

POIROT is accessible at https://doi.org/10.5281/zenodo.15586319 or https://github.com/HexHive/droidot.git.

A.2.3 Hardware dependencies

Our evaluation requires Android emulators to run the mobile applications under test. Modern Android emulators require ARM64 architecture for optimal performance and compatibility with contemporary Android versions.

To accommodate reviewers during the artifact evaluation process, we provide access to two dedicated servers. The first one is an x64 server, `ssh -p 1234 ae@65.108.89.50`, which is used to run the static analysis tools for the first major claim of our paper (C1).

The second one is an ARM64 server, `ssh -p 1235 ae@65.108.89.50`, which is used to run the fuzzing campaigns for the second major claim of our paper (C2) and to reproduce the case study crash for the third major claim of our paper (C3).

This server enables reviewers to reproduce our experimental results without requiring them to have ARM64 hardware locally. However, we note that those servers have more limited computational resources compared to the AWS cluster infrastructure used in our original paper evaluation.

The difference in computational resources between the evaluation servers and our original experimental setup should not affect any of the major claims made in our paper.

A.2.4 Software dependencies

All software dependencies are managed through our Docker container.

The Docker container ensures reproducible builds and eliminates dependency conflicts across different host systems. No additional software installation is required on the host machine beyond Docker itself.

A.2.5 Benchmarks

As outlined in the paper, there are no clear benchmarks defined by literature to evaluate Android native library fuzzers. Therefore, included in the artifact we ship the most 100 popular Android applications. They can be found in the `target_APK` folder.

Note: we kindly ask the reviewers to not redistribute the APKs, as they are sourced from Androzoo and are subject to their terms of use.

A.3 Set-up

A.3.1 Installation

To access the artifact servers, please provide your SSH public key in the HotCRP comments.

ARM64 Server for Fuzzing Campaigns:

1. Login to the server (`ssh -p 1235 ae@65.108.89.50`).

2. Clone the POIROT repository: `git clone https://github.com/HexHive/droidot.git`.
3. Change to the repository directory: `cd droidot`.
4. Run `./setup.sh` to download dependencies and build the Docker container.
5. Spawn a shell in the container: `./run.sh`.

x86 Server for Static Analysis Tools:

1. Login to the server (`ssh -p 1234 ae@65.108.89.50`).
2. Clone the POIROT repository: `git clone https://github.com/HexHive/droidot.git`.
3. Change to the repository directory: `cd droidot`.
4. Change to the static analysis AE directory: `cd static_ae`.
5. Run `./setup.sh` to download dependencies and build the Docker container.
6. Run `./build_docker.sh` to download dependencies and build the Docker container.
7. Spawn a shell in the container: `./start_docker.sh`.

A.3.2 Basic Test

ARM64 Fuzzing Server
(`ssh -p 1235 ae@65.108.89.50`)

Run the `./start_single_emu.sh` script. After the script has finished running `adb devices` should show at least one emulator, `emulator-5554`, inidicating that the emulator was started succesfully.

x86 Static Analysis Server
(`ssh -p 1234 ae@65.108.89.50`)

Run the `./basic_test.sh` script. The script will analyze an app with FlowDroid. After a couple of minutes, the script should print `Found 2 leaks`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Android dataflow state of the art tools do not scale to large apks. POIROT's analysis passes scale to large apks.
(C2): POIROT's analysis passes positively impact coverage.
(C3): POIROT finds bugs in real-world apps.

A.4.2 Experiments

The first experiment will be run on the x86 server. The second and third experiments will be run on the ARM64 server.

(E1): [C1] [10 human-minutes + 10 compute-hour]:
Preparation: Login to the x86 server (`ssh -p 1234 ae@65.108.89.50`) and open the script `run_rq1.sh` in an editor to edit the `NO_OF_APPS` variable. In our paper we used 100 apps, but with the timeout of 30 minutes and testing 5 tools, that amounts to 0.5h × 100 × 5 = 250 hours. For the artifact evaluation we limited the `NO_OF_APPS` variable to 10 apps. We believe this reduced dataset is still representative of the original results.
In the folder `target_APK` we provide the 100 APKs used in the paper (the most popular 100 apps) However, reviewers are also welcome to test with different apps. To do so, they can add new apps to the `target_APK` folder, with both the folder and the apk name as the package name of the app (e.g. `com.example.app/com.example.app.apk`).
Execution: Start a tmux session with `tmux` and run the script `run_rq1.sh`.
Results: The script prints the result of the various tools and the failure symptom. The results should be congruent with Table 2.

(E2): [C2] [10 human-minutes + 12 compute-hour]: Fuzzing ablation experiment on POIROT's analysis passes.
Preparation: Login to the arm64 server (`ssh -p 1235 ae@65.108.89.50`), start a tmux session with `tmux` and then start the container with `./run.sh` .
Execution: Run the `./run-ablation.sh` script. The script runs the ablation study fuzzing campaign, fuzzing a subset of the apks from the dataset with /without the argument value and call sequence analysis pass.
Results: After finishing the script generates the plots from Figure 5 and Figure 6 in the output-dir: `$(datetime)_fuzzing_data`. Download them from the server: `scp -r -P 1235 ae@65.108.89.50:/home/ae/droidot/[output-dir]` and inspect the pdfs. These should show that enabling analysis passes results in coverage increase. (larger green area).

(E3): [C3] [5 human-minutes + 20 compute-minutes]: Reproduce the case study crash.
Preparation: Login to the arm64 server (`ssh -p 1235 ae@65.108.89.50`), start a tmux session with `tmux` and then start the container with `./run.sh`.
Execution: Run the `./reproduce-tplink.sh` script.
Results: The script generates the harness for the vulnerable function, fuzzes it and then reproduces the crash. After finishing the script prints the backtrace of the reproduced and deduplicated crash. It should look similar to this:

```
abort
scudo::die
scudo::ScopedErrorReport::~ScopedErrorReport
scudo::reportInvalidChunkState
scudo::Allocator<scudo::AndroidConfig, &scudo__malloc__postinit>::deallocate
mp4__write__one__jpeg
Java__com__tplink__skylight__common__jni__MP4Encoder__packVideo
fuzz__one__input
main
```

The backtrace contains the allocators detection of double free indicating the detection of the use after free.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: No Way to Sign Out? Unpacking Non-Compliance with Google Play's App Account Deletion Requirements

Jingwen Yan[†], Song Liao[‡], Jin Ma[†], Mohammed Aldeen[†], Salish Kumar[‡], Long Cheng[†]
[†]Clemson University, {jingwey, jin7, mshujaa, lcheng2}@clemson.edu
[‡]Texas Tech University, {song.liao, salikuma}@ttu.edu

A Artifact Appendix

A.1 Abstract

In this work, we developed a tool named DELETETRACKER to semi-automatically detect various account deletion issues as defined in the paper. DELETETRACKER comprises four modules: Account Deletion Link Crawler, Account Deletion Link Analyzer, Account Deletion Method Classifier, and Account Deletion Validator. The Account Deletion Link Crawler automatically collects account deletion information from both the Google Play mobile app and website, preparing for subsequent analysis. Next, the Account Deletion Link Analyzer examines the availability, cross-platform inconsistency, accessibility, and similarity of account deletion links. The Account Deletion Method Classifier detects content issues, including missing account deletion method, alternative account deletion method, missing web-based account deletion method, and missing in-app path account deletion method. Finally, the Account Deletion Validator verifies whether users can successfully delete their accounts using the methods provided in the apps' account deletion links. Since the Account Deletion Verifier is performed manually, we do not include this module in the artifact. This artifact is intended to demonstrate the functionality of the first three modules of DELETETRACKER, as described in Sections 4.1, 4.2, and 4.3 of our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact does not execute destructive actions, modify system settings, or disable any security mechanisms. However, we would like to note the following considerations:

- **Crawler Rate Limiting:** When using the Account Deletion Link Crawler, evaluators should avoid sending an excessive number of requests in a short period of time. High-frequency scraping may trigger rate limiting or temporary access blocks by Google.
- **OpenAI API Key Usage:** If evaluators use the provided OpenAI API key for the Account Deletion Method Classifier, care must be taken not to expose or misuse the key. The key is intended for demonstration purposes only.

A.2.2 How to access

The artifact is stored in the Zenodo and the DOI link is https://doi.org/10.5281/zenodo.15610829. To obtain the artifact, please download the files using the URL provided above.

A.2.3 Hardware dependencies

While our artifact does not require specialized hardware, we recommend running the Account Deletion Link Crawler on a system with a quad-core processor, 16 GB RAM, and an SSD to ensure smooth and efficient performance when using the Android Emulator and Appium.

A.2.4 Software dependencies

The artifact requires a system running macOS, which was used as our test environment. We have not tested the artifact on other operating systems and therefore do not claim cross-platform compatibility.

The following software components are required:

- **Docker**: Used to ensure consistent environment setup and execution for part of the provided code.
- **Android Studio**: Required for running the Android Emulator.
- **Appium**: Required to automate interactions with the Android Emulator and Google Play Store.
- **Google Account**: A Google account is required to log in to the emulator's Google Play Store during execution.
- **OpenAI API Key**: Required to run the GPT-based account deletion method classifier. A valid key has been provided through the HotCRP system.

A.2.5 Benchmarks

We have included all necessary data in the `/data` and `/artifact/result` directories. The `/data` directory contains 50 apps used in the pilot study, 863 apps used during the development of DELETETRACKER, and 100 apps used for end-to-end performance evaluation. The `/artifact/result` directory includes 10 apps used to test the mobile crawler and 100 apps used to evaluate all other modules.

A.3 Set-up

A.3.1 Installation

To install the required dependencies and set up the artifact environment, please follow the steps below:

1. Download the repository from https://doi.org/10.5281/zenodo.15610829 and enter the artifact directory.
2. Install Python dependencies: pip install -r requirements.txt
3. Install spaCy language model: python -m spacy download en_core_web_sm
4. Install Docker: Docker can be downloaded and installed from https://www.docker.com/.
5. Install Android Studio and set up an emulator (Android Studio can be downloaded and installed from https://developer.android.com/):
   - Open Android Studio and go to "More Actions > Virtual Device Manager".
   - Create and launch a virtual device (e.g., Pixel 9) with Google Play support.
   - Log into Google Play using a valid Google account.
   - Ensure the emulator has Chrome installed.
6. Install node: brew install node.
7. Install and start Appium. Appium can be downloaded and installed from https://appium.io/docs/en/2.2/quickstart/install/. After installation, go to the terminal and start Appium by using the command `appium`.
8. Install the driver: appium driver install uiautomator2.
9. ADB Setup and Emulator Verification: To run the mobile crawler or interact with Android emulators, you must ensure that adb (Android Debug Bridge) is installed and properly configured in your environment.
   - If adb is not recognized in your terminal (zsh: command not found: adb), add the following lines to your /.zshrc: `exportANDROID_HOME=$HOME/Library/Android/sdk;exportPATH=$ANDROID_HOME/emulator:$ANDROID_HOME/tools:$ANDROID_HOME/tools/bin:$ANDROID_HOME/platform-tools:$PATH`.
   - Reload the configuration: source /.zshrc.
   - Verify ADB Connection: adb devices
10. Set OpenAI API key: Use the key provided through HotCRP and replace the API key in Line 13 of artifact/3_AccountDeletionMethodClassifier/GPT_classifier.py.

After completing these steps, the evaluator should be able to execute the core functionality described in Sections 4.1–4.3 of the paper.

A.3.2 Basic Test

To verify that all required components are correctly installed and working, we provide a simple functionality test. This test focuses on the Mobile Account Deletion Link Crawler module, as all other modules execute quickly and do not require additional setup. This test verifies the integration of Appium, the Android Emulator, and access to the Google Play Store. We provide a list of 10 apps in the `/artifact/result` directory for testing. Evaluators can run the crawler on these apps using the following file: `artifact/1_AccountDeletionLinkCrawler/1_mobile_crawler.py`. The script will attempt to retrieve account deletion-related links from Google Play for the given apps. The output will be saved to `artifact/result/10_test_app_link.csv`. To assist evaluators, we also provide an example video demonstrating how to run the crawler and visualize the step-by-step process for these 10 apps.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): DELETETRACKER can automatically crawl and collect account deletion links from Google Play mobile app and webpage using both mobile and web crawlers. This is proven by the experiments (E1).

(C2): DELETETRACKER can identify multiple types of link-related issues, including inaccessible, inconsistent, and duplicate account deletion links across platforms. This is proven by the experiments (E2).

(C3): DELETETRACKER can classify the account deletion method using a combination of NLP and GPT-based analysis. This is proven by the experiments (E3).

A.4.2 Experiments

(E1): *[Account Deletion Link Crawler] [30 human-minutes]*
Preparation: Please refer to Section A.3.1 to prepare virtual device setup and start appium Server.
Execution: Enter `1_AccountDeletionLinkCrawler`. First, run `1_mobile_crawler.py`, then run `2_web_crawler.py`.
Results: After both scripts have been executed, check the updated CSV file in the `artifact/result` directory. The mobile crawler will add two new columns: `delete_account_url` and `manage_data_url`. The web crawler will add one new column: `Data Deletion Info`.

(E2): *[Account Deletion Link Analyzer] [10 human-minutes]*
Preparation: Please refer to Section A.3.1 to prepare required Python dependencies.
Execution: Run all the scripts in the `2_AccountDeletionLinkAnalyzer` directory (the execution order does not matter).
Results: After all scripts have been executed, check the updated CSV file in the `artifact/result` directory. Three new columns will be generated in the result CSV file: `delete_account_url_available`, `inconsistency`, and `domain`, corresponding to different types of link issues.

(E3): *[Account Deletion Method Classifier] [30 human-minutes]*
Preparation: Please refer to Section A.3.1 to prepare required Python dependencies.
Execution: For `3_AccountDeletionMethodClassifier`, please first run `web2html.py` to convert URLs into HTML files. Then, enter the `HtmlToPlaintext-master` folder and follow the `README` instructions to build and run the Docker container using `./build.sh` and `./run.sh`. The remaining scripts have no specific execution order and can be run independently after preprocessing is complete.
Results: After `web2html.py` and the HtmlToPlaintext tool have been executed, please check the `artifact/3_AccountDeletionMethodClassifier/HtmlToPlaintext-master/ext` directory to ensure it contains both HTML files and corresponding plaintext files. After running all the remaining classification scripts, six new columns will be generated in the result CSV file: `contains_delete`, `url_contains_delete`, `has_user_input_form`, `has_delete_button`, `GPT_Classification`, and `StepbyStep`, representing a different classification result.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Lost in the Mists of Time: Expirations in DNS Footprints of Mobile Apps

Johnny So
Stony Brook University

Iskander Sanchez-Rola
Norton Research Group

Nick Nikiforakis
Stony Brook University

A Artifact Appendix

This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of your artifact. It should include a clear description of the hardware, software, and configuration requirements. In case your artifact aims to receive the functional or results reproduced badge, it should also include the major claims made by your paper and instructions on how to reproduce each claim through your artifact. Linking the claims of your paper to the artifact is a necessary step that ultimately allows artifact evaluators to reproduce your results.

Please fill all the mandatory sections, keeping their titles and organization but removing the current illustrative content, and remove the optional sections where those do not apply to your artifact.

A.1 Abstract

In this work, we present the first large-scale analysis of mobile app dependencies through a dual perspective accounting for time and version updates, with a focus on expired domains. First, we detail a methodology to build a representative corpus comprising 77,206 versions of 15,124 unique Android apps. Next, we extract the unique eTLD+1 domain dependencies — the "DNS footprint" — of each APK by monitoring the network traffic produced with a dynamic, UI-guided test input generator and report on the footprint of a typical app. Using these footprints, combined with a methodology that deduces potential periods of vulnerability for individual APKs by leveraging passive DNS, we characterize how apps may have been affected by expired domains throughout time. Our findings indicate that the threat of expired domains in app dependencies is nontrivial at scale, affecting hundreds of apps and thousands of APKs, occasionally affecting apps that rank within the top ten of their categories, apps that have hundreds of millions of downloads, or apps that were the latest version. Furthermore, we uncovered 40 immediately registrable domains that were found in app footprints during our analyses, and provide evidence in the form of case studies as to their potential for abuse. We also find that even the most security-conscious users cannot protect themselves against the risk of their using an app that has an expired dependency, even if they can update their apps instantaneously.

As part of the artifact evaluation, we release datasets and analysis notebooks that enable reviewers to reproduce the figures and tables that are presented in the text. Additionally, we release a version of the main app analysis infrastructure to enable future work.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, and ethical concerns associated with running these artifacts. In terms of running processes, running the Jupyter notebooks requires a running Jupyter server, and running the Cuttlefish infrastructure spawns various disposable components. However, we do recommend using a local firewall (e.g., ufw) if running the Cuttlefish infrastructure, as it will spawn Cuttlefish virtual devices that can be manipulated over the network via adb. The only host-level settings that are modified come from the Cuttlefish infrastructure, which provides scripts to manipulate iptables and ufw rules (`iptables.sh` and `ufw_modify_cvd.sh` in the `/cuttlefish/scripts` directory, respectively). These scripts enable all traffic from the Cuttlefish virtual devices to correctly pass through ufw (if it is enabled), and route DNS traffic from the Cuttlefish devices to a specified DNS server.

A.2.2 How to access

We provide access to our artifact on Zenodo at the following link: https://doi.org/10.5281/zenodo.14737144. The artifact will be updated with new versions at this URL according to the discussion of the artifact evaluation period.

A.2.3 Hardware dependencies

The only hardware requirements are imposed by the Android Cuttlefish infrastructure components, which require CPU support for `kvm`.

For convenience, we provide reviewers with access to a VM which we have pre-configured with all necessary dependencies, and installed the artifacts at `/home/ubuntu/artifacts`. In addition, this VM comes with the sample of APKs used for the Cuttlefish experiments

(which unfortunately cannot be publicly shared in the artifact itself). We recommend using the provided VM as a reference as there are various dependencies required by the Cuttlefish infrastructure components.

A.2.4 Software dependencies

These artifacts were exclusively developed and tested on an Ubuntu 20.04 machine. The Jupyter notebooks should work on any platforms that support Python, and the Cuttlefish-based infrastructure should work on most Linux distributions. If you would like to run the analysis notebooks, the dependencies are fairly simple. However, if you would like to set up your own testing environment with the Cuttlefish virtual devices, there are additional dependencies that need to be installed, and certain components may not be cross-platform compatible (e.g., the script that uses `iptables` to route DNS traffic from Cuttlefish devices).

To run the Jupyter Notebooks, the only requirement is a Python environment with all the dependencies in `/requirements.txt`.

To run the Cuttlefish app analysis infrastructure, there are additional requirements:

1. Docker Engine for the DNS servers and HTTPS proxies
2. Android Cuttlefish - which can be installed using `cuttlefish_setup.sh` and `cuttlefish_download_images.sh` in the `/cuttlefish/scripts` directory
    - **NOTE:** `kvm` needs to be supported by your CPU
3. The fork of DroidBot in `droidbot.tar.gz` installed in the Python environment
    - After inflating the `droidbot` folder, it can be installed via `pip install -e /path/to/droidbot`
4. Frida server - which can be installed by `download_frida_server.sh` in the `/cuttlefish/scripts` directory
5. `iptables` to route DNS traffic from the Cuttlefish devices
6. **[recommended]** `ufw` as Cuttlefish devices expose additional ports
7. **[recommended]** a desktop environment / VNC server (e.g., `turbovnc`) to interact with the Cuttlefish devices

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

As Zenodo records only support flat files, the artifacts have been compressed.

1. Download the data artifacts from Zenodo at https://doi.org/10.5281/zenodo.14737144 and ensure they are all in the same directory.
2. `chmod +x inflate.sh && chmod +x deflate.sh`
3. `/inflate.sh` to inflate the artifacts

Depending on whether you would like to only run the analyses with Python or run the Cuttlefish infrastructure, you will need to install different dependencies. The steps below outline the installation steps for both cases.

Python Environment [Jupyter and Cuttlefish]. The official downloads can be found on the Python site, but Python is likely to have been pre-installed on your machine. We recommend creating a separate virtual environment for the required Python dependencies for this artifact (see this for a primer on virtual environments).

1. After setting up your Python virtual environment, please install the packages in `/requirements.txt` with the corresponding commands for your environment manager (e.g., `pip install requirements.txt`).

The below dependencies are only required to test the Cuttlefish infrastructure.

Python Environment [Cuttlefish, required]. After installing your Python environment as described above, you will also need to install the provided fork of DroidBot to use in the Cuttlefish infrastructure. To do so, please inflate the artifacts, and run:

```
pip install -e /path/to/droidbot-fork
```

inside your Python virtual environment.

Docker [Cuttlefish, required]. The official setup instructions can be found at the Docker docs site.

1. Docker Engine can be installed on Linux with `docker_setup.sh` in the `/cuttlefish/scripts` directory.
2. Log out and log back in so that your group membership is re-evaluated to run `docker` commands without `sudo`.

Android Cuttlefish [Cuttlefish, required]. The official setup instructions can be found at the Android Open Source site.

1. Cuttlefish packages can be set up on Linux with `cuttlefish_setup.sh` in the scripts directory.

2. Download the following Cuttlefish Android 11 artifacts from aosp-android11-gsi@11718355 to some directory (e.g., `$HOME/cf-images/11718355` which is created by the prior script):
   (a) `aosp_cf_x86_64_phone-img-11718355.zip` for the Android image
   (b) `cvd-host_package.tar.gz` for the host cuttlefish utilities
3. Extract the downloaded artifacts by running:

```
tar -xvf cvd-host_package.tar.gz
unzip aosp_cf_x86_64_phone-img-11718355.zip
```

4. Then, add the absolute path of the `cf-images/android11/bin` to your PATH (it is recommended to add to do so in a persistent manner)

Desktop Environment/VNC [Cuttlefish, recommended]. If you are working in a remote/headless environment, it is recommended to install a desktop environment and a VNC server so that you can visually monitor and control the Cuttlefish virtual devices with a browser and WebRTC.

It should be possible to connect to a remote WebRTC process from your local computer without installing a desktop environment on the machine running the Cuttlefish infrastructure (e.g., with SSH port tunneling), but we encountered problems with this and found that installing the desktop environment was simpler.

If you need help, here is an example guide on how to install TightVNC with Xfce4 on Ubuntu.

Configuration. After installing the required dependencies, make sure to update the configuration for the desired components. For the Jupyter notebooks, the main configuration file is `/analysis/.parameters.py`, but this should not require any changes as long as the directory structure was not modified after inflating the artifacts.

For the Cuttlefish infrastructure, the main configuration file is `/cuttlefish/.env`. In particular, make sure to change the following variables:

1. `DIR_BASE` to the absolute file path to the inflated `/data/cuttlefish` directory
2. `DIR_ADB_FILES` to the absolute file path to the inflated `/cuttlefish/adb_files` directory
3. `MITMPROXY_CACERT_FILENAME` to be the name of the created mitmproxy CA certificate produced by the `generate_mitmproxy_cert.sh` script. See Section A.3.2 for more details on what to put for this setting.
4. `NUM_APPS_PER_SAMPLE` to the number of apps per sample group desired
5. `NUM_CUTTLEFISH_DEVICES` to the number of Cuttlefish devices

Additionally, configure `sudo` to allow executing the script `cuttlefish/scripts/iptables.sh` without requiring a password. This script is executed for each Cuttlefish virtual device in the current user, so it normally requires elevated privileges. After verifying the contents of the script, you can do this by running `sudo visudo` and adding the following line:

```
your_username ALL=(ALL) NOPASSWD: /path/to/cuttlefish/scripts/iptables.sh
```

A.3.2 Basic Test

To test functionality of the Jupyter notebooks, activate the Python environment (with `poetry shell` in the artifact directory in the provided VM), and launch a Jupyter server by running `jupyter lab` in the artifact directory. Then, configure SSH port forwarding to your local machine over the default Jupyter port 8888, and navigate to the URL from the output of the Jupyter command in your browser of choice.

To test functionality of the Cuttlefish devices after installing all dependencies, perform the following:

1. Navigate to your `cf-images` directory where you downloaded the Android Cuttlefish images (e.g., `$HOME/cf-images/11718355`), and run the following:
   `HOME=$PWD ./bin/launch_cvd -num_instances=2 -resume=false -start_webrtc=true -start_vnc_server=false`
2. Wait until you see a message in the output (colored green) asking you to navigate to `https://localhost:8443`
3. Connect to the desktop environment, launch a browser, and navigate to `https://localhost:8443`.

Next, bootstrap the mitmproxy CA certificate creation that will be injected into the Android devices by doing the following:

1. In `/cuttlefish/compose.yaml`, change the file path of the `hardumps` volume to the absolute path of the inflated `data/cuttlefish/hardumps` folder.
2. Navigate to the `/cuttlefish` directory and run `docker compose up -build -d`.
3. After the containers have been created, run the `generate_mitmproxy_cert.sh` script from the same directory.
4. The CA certificate should then be in the `/cuttlefish/adb_files` directory (i.e., `c8750f0d.0`)

5. Make sure to change the environment variable `MITMPROXY_CACERT_FILENAME` in `/cuttlefish/.env` to the name of the certificate.

Then, test that the Cuttlefish virtual device is appropriately configured by the infrastructure by running `python analyze.py -test`. You should see that the Cuttlefish devices are set up before the script exits.

Finally, modify the input APKs to run with the Cuttlefish infrastructure by:

1. Modifying the file `samples_cuttlefish.csv` in `/data/cuttlefish/samples`, adding the SHA256 `hash`, `package` name, `version` string, and `group` name (the provided samples file uses `successful` and `unsuccessful` to refer to the originally-successful and originally-unsuccessful APKs).

2. Place the added APKs into the directory `DIR_BASE/samples/<group>` with the name `<package>_<version>_<hash>.apk`.

3. To run the actual analysis, run `analyze.py` without the test flag.

A.4 Evaluation workflow

A.4.1 Major Claims

The major claims made in the paper are as follows:

(C1): *Expired domain names are found in the DNS traffic of multiple versions of Android apps, regardless if they are out of date or the latest available versions.*

(C2): *The purpose and use of many such domains can be identified from the context of the request and inspecting the decompiled APKs, and they can be abused by malicious re-registrants to change app behavior, even if the app itself does not change.*

(C3): *All eight figures and four tables presented in the text can be produced from our extracted data. However, please note that because the commercial telemetry data we used cannot be released, this artifact produces slightly different versions of Figures 3 and 4, and Table 4. Additionally, Figure 7 is also slightly different, but because of minor differences in how the original code processed the raw data.*

(C4): *The app analysis infrastructure extracts DNS network traffic of APKs.*

A.4.2 Experiments

The following experiments can be performed to verify that the artifacts are functional and can be used to reproduce the results from the text. The Jupyter notebooks used for analysis are bundled with pre-processed data so that the raw data — which is large — does not have to be re-processed.

(E1): *[5 human-minutes + 5 compute-minutes + 0GB disk]: Run all cells in the footprints.ipynb notebook and verify that domains were checked for expirations.*
How to: *Ensure the basic test for the Jupyter notebooks has been completed, with a Jupyter server now running in a Python environment with the dependencies installed.*
Preparation: *Connect to the Jupyter server with your browser and open the `analysis/footprints.ipynb` notebook.*
Execution: *Press Run → Run All Cells.*
Results: *The `df_footprints` variable is a DataFrame that has a column named `expired_at_exec`, denoting domains that were expired during app execution. The rows that have this column as True span multiple versions of different apps.*

(E2): *[30 human-minutes + 5 compute-minutes + 0GB disk]: Run all cells in the decompilations.ipynb notebook and verify that most of the domains can be found directly inside the APK, and their purpose can be identified from inspecting the decompiled APKs.*
How to: *Ensure the basic test for the Jupyter notebooks has been completed, with a Jupyter server now running in a Python environment with the dependencies installed.*
Preparation: *Connect to the Jupyter server with your browser and open the `analysis/decompilations.ipynb` notebook.*
Execution: *Press Run → Run All Cells.*
Results: *Verify that the `Searching for Domains in Decompiled Code` launches grep commands to search the decompiled APKs for their corresponding domains. Manually confirm that files in the directory `DIR_BASE → samples-jadx-processing/expired_at_reg` contain the output of the grep commands. Randomly sample some of the identified domains to look at the decompiled APKs.*

(E3): *[10 human-minutes + 10 compute-minutes + 0GB disk]: Run all cells in each Jupyter notebook and verify that all figures and tables are produced.*
How to: *Ensure the basic test for the Jupyter notebooks has been completed, with a Jupyter server now running in a Python environment with the dependencies installed.*
Preparation: *Connect to the Jupyter server with your browser and open the `footprints.ipynb`, `decompilations.ipynb`, and `cuttlefish.ipynb` notebooks in the `analysis/` directory.*
Execution: *Press Run → Run All Cells in each notebook.*
Results: *Verify that `footprints.ipynb` produces Tables 1, 2, and 4 and Figures 2, 3, 4, 7, and 8, `decompilations.ipynb` produces Tables 3 and 4, and Figure 5, and `cuttlefish.ipynb` produces Figure 6.*

(E4): *[10 human-minutes + 3*2*N/D compute-minutes + 0GB disk]: Launch the Cuttlefish infrastructure on a sample of APKs.*
How to: *Ensure the basic test for the Cuttlefish infrastructure setup has been completed and the configuration has been updated.*
Preparation: *Modify the* `cuttlefish/.env` *file and change* `NUM_APPS_PER_SAMPLE`*, the number of apps that will be analyzed from each of the two sample groups (N), and* `NUM_CUTTLEFISH_DEVICES`*, the number of live Cuttlefish virtual devices (D).*
Execution: *Using the configured Python environment, run* `python analyze.py` *in the* `/cuttlefish` *directory.*
Results: *When analysis of an app completes, there is a pcap file in* `DIR_BASE` → `DIR_DATADUMPS` → `tcpdumps` → `<group>` → `<package>_<version>_<hash>_<timestamp>.pcap`.

A.5 Notes on Reusability

The number of Cuttlefish devices that can be launched concurrently can be adjusted by modifying the value of the `num_instances` flag provided to the `launch_cvd` command, changing the `NUM_CUTTLEFISH_DEVICES` variable in `cuttlefish/.env`, and adding additional Docker containers to `cuttlefish/compose.yaml` if necessary. Furthermore, the Cuttlefish infrastructure can be reused for different APK inputs, and can be done by modifying the input `samples_cuttlefish.csv` file that describe the APKs and placing them into the expected locations. The analysis notebooks will continue to function even with the addition of new data, although it may also be desirable to insert your own API keys for Farsight and Dynadot to analyze new domains.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: TapTrap: Animation-Driven Tapjacking on Android

Philipp Beer
TU Wien

Marco Squarcina
TU Wien

Sebastian Roth
University of Bayreuth

Martina Lindorfer
TU Wien

A Artifact Appendix

A.1 Abstract

We provide multiple artifacts to reproduce the results presented in the paper and support future work that builds on our findings. Specifically, we include the following artifacts:

- **Dataset preparation:** Scripts for crawling the Google Play Store, downloading apps, and preparing them for analysis (`/dataset_preparation`). Due to the dataset size, we cannot publicly release the full set of apps used. Reviewers are granted access to it as outlined in Section A.3.1.
- **Malicious app detection:** Code and results for malicious app detection (`/malicious_app_detection`).
- **Vulnerable app detection:** Code and results for vulnerable app detection (`/vulnerable_app_detection`).
- **User study:** Materials from the user study, including the information sheet, consent forms, questionnaires, app, and website used during the study (`/user_study`).
- **TapTrap PoC:** Proof-of-concept of TapTrap (`/poc`).
- **Reproducibility scripts:** Scripts to reproduce the results presented in the paper (`/reproducibility`).
- **Supplementary files:** Additional files not directly relevant for artifact evaluation, such as assets and paper-related licenses (`/assets` and `/paper_licenses`)

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Excessive or rapid scraping and downloads of APKs from the Play Store may violate the Play Store's terms of service and could result in temporary or permanent IP bans. The dataset preparation pipeline uses a conservative approach to avoid excessive downloads. We nevertheless recommend reviewers use a dedicated machine to avoid potential issues.

Running the KillTheBugs app and the TapTrap PoC are intended solely to demonstrate the security vulnerability described in the paper. We do not perform destructive operations and do not collect personal data.

A.2.2 How to access

The artifacts accompanying the paper are hosted at https://github.com/beerphilipp/taptrap and archived at https://doi.org/10.5281/zenodo.15519676.

A.2.3 Hardware dependencies

Access to the APKs. Due to the size of the APK dataset used in the paper, we cannot distribute it directly. Reviewers are granted access to it as outlined in Section A.3.1.

System Requirements. Both x86 and ARM architectures are supported. For running an Android emulator, however, we recommend a Mac with Apple Silicon. A minimum of 16 GB RAM and 150 GB available disk space are suggested.

Physical Android Device. We recommend running the apps on a physical Pixel 6a device running Android 15. While they can be run on other devices, screen element positioning may differ and require manual adjustment for the attack to run. Alternatively, the app can be executed in an emulator.

A.2.4 Software dependencies

Operating System. We have tested and support Ubuntu 24.04 and macOS 15 with a desktop environment. Other systems *may* require adjustments.

Docker. Install Docker (see `/reproducibility/README.md` for a step-by-step guide or https://docker.com/get-started for official instructions).

Rsync. We require `rsync` (preinstalled on Ubuntu 24.04 and macOS 15) to retrieve the APK dataset. We have, however, experienced issues with it on macOS 15 and suggest installing it via Homebrew instead (`brew install rsync`).

Java. To be able to use the necessary Android dependencies, install a recent version of Java (see https://www.java.com/en/download/manual.jsp)

Android Dependencies. Install the Android dependencies (see `/reproducibility/README.md` for a step-by-step guide):

- Download and install the Android command line tools, which include `sdkmanager`.
- Install the platform tools to install `ADB`.
- Set the `ANDROID_HOME` environment variable.
- Add the platform tools to `$PATH`.

A.2.5 Benchmarks

Evaluating the analyses pipelines requires access to the APK dataset we used. See Section A.3.1 for access instructions.

A.3 Set-up

A.3.1 Installation

Clone the Repository. Clone the artifact repository using `git clone https://github.com/beerphilipp/taptrap.git` in a directory of your choice.

Install the Dependencies. See A.2.4 for instructions.

Obtain a Google AAS token. Downloading apps from the Play Store requires a Google account and an AAS token. We provide such credentials for reviewers to use. Other researchers may create a new Google account and refer to `/dataset_preparation/downloader/README.md` for a summary on how to generate an AAS token.

Download APKs. Our experiments use a subset of 500 randomly selected apps and 266 predefined apps from the dataset. To access the dataset, save the private SSH key that we provide for reviewers to a file named `~/.ssh/taptrap_key` and give it the correct permissions with `chmod 600 ~/.ssh/taptrap_key`. Researchers may request access to the APK dataset by contacting the authors.
Run the following commands in the artifact's root directory:

- Select 500 random apps:

```
rsync -e "ssh -i ~/.ssh/taptrap_key" -azn \
--out-format="%n" dl@download.st1.secpriv.wien: . | \
grep -v "/$" | sort -R | head -n 500 > /tmp/apps.txt
```

- Add the set of predefined apps:

```
cat reproducibility/fixed_apps.txt >> /tmp/apps.txt
```

- Download the selected apps, where `<DIR>` refers to where the APKs should be stored:

```
rsync -e "ssh -i ~/.ssh/taptrap_key" -avxz \
--files-from /tmp/apps.txt \
dl@download.st1.secpriv.wien: <DIR>
```

Start the Android Emulator or Connect a Device. Connect the physical device to the host machine via USB. If you are using an emulator, run `reproducibility/start_emulator.sh` in the repository's root directory to automatically download the correct emulator image for your system and start it.

Set up the Android device. On physical devices, enable USB debugging in *Settings* > *About phone*, then tap *Build number* seven times to enable developer options. Go to *System* > *Developer options* and enable *USB debugging*. This step is not required for emulators.

A.3.2 Basic Test

To verify correct installation and setup, run `reproducibility/basic_test.sh <email> <token>` in the repository's root directory. Replace `<email>` and `<token>` with the Google credentials. This command will perform the following steps:

- Builds all required Docker images.
- Checks if the provided Google credentials are valid by attempting to download an app.
- Checks if an Android device is connected.

The script should print *OK* to the console. Depending on the device resources, this may take up to 20 minutes.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The app downloading and preparation process from the Play Store, as described in Section 5.1, is functional and yields a large dataset for further analysis. This is proven by experiment *E1*.

(C2): TapTrap is effective on Android 15 as described in Section 3 of the paper. This is proven by experiment *E2*.

(C3): A large-scale analysis of 99,705 apps using the animation detection pipeline (cf. Section 5.2) identified 61 unique animations that bypassed the intended duration limit of 3,000ms (Section 5.3.1) and 28 apps containing animations with a maliciousness score of at least 50 (cf. Section 5.3.2). This is proven by experiment *E3*.

(C4): 76.3% of analyzed apps are vulnerable to TapTrap based on the static analysis pipeline provided in Section 6 and Table 3. This is proven by experiment *E4*.

A.4.2 Experiments

(E1): Dataset preparation [15 human-minutes + 1 compute-hour + 20 GB disk]

Preparation: Follow the steps in Section A.3.1 to set up the environment.

Execution: Run `reproducibility/e1.sh <EMAIL> <TOKEN> <OUT>` in the repository and replace `<EMAIL>` with the Google account email, `<TOKEN>` with the AAS token, and `<OUT>` with the desired output directory. The script performs a Play Store crawl and download and prepares the APKs for analysis. Due to time constraints, the crawl stops after 25,000 package names, and only a random subset of 50 apps is attempted for download.

Results: The script should output `OK`.

Alternatively, manually verify that:

- `<OUT>/apps.csv` contains $\geq 15,000$ package names (lower bound after crawling 25,000 package names)
- `<OUT>/apps` contains ≥ 30 apps (lower bound after downloading 50 apps)
- Note that lower bounds are due to regional restrictions and possible device limitations.

(E2): TapTrap functionality [30 human-minutes + 10 compute-minutes + 15 GB disk]

Preparation: Follow the steps in Section A.3.1 to set up the environment, then run `reproducibility/e2.sh` from the repository root to install and launch the app. Note that running an emulator on x86 with nested virtualization is extremely slow and may cause the attack to fail.

Execution: Click the "Start" button in the app to initiate the attack, then click "Click here". The app secretly opens a camera permission prompt and attempts to trick the user into granting access.

Results: The app should display *"Permission granted"* without the user being aware of granting the permission.

Alternatively, manually verify that camera access has been granted to the app: long-press the app icon, select "App info", then "Permissions".

(E3): Detection of potentially malicious animations [15 human-minutes + 45 compute-minutes + 100 GB disk]:

Preparation: Follow the steps in Section A.3.1 to set up the environment, including downloading the APKs.

Execution: Run `reproducibility/e3.sh <APK_DIR> <OUT_DIR>` in the repository root, where `<APK_DIR>` is the input directory containing the APKs and `<OUT_DIR>` is the output directory for results. The script analyzes a random subset of 500 apps, plus a fixed subset of apps that span the 61 animations exceeding 3,000 ms in duration and those that have a score of at least 50.

Results: The script should output `OK`.

Alternatively, manually verify that the generated report at `<OUT_DIR>/report.tex` states:

- `maltapNumberUniqueAnimationsExtendedDuration`: 61 unique animations exceeding the 3,000 ms duration threshold were found (cf. Section 5.3.1).
- `maltapNumberAppsAnimationsScoreMin`: 28 apps containing at least one animation with a maliciousness score of at least 50 were found (cf. Section 5.3.2).

(E4): Detection of vulnerable apps [15 human-minutes + 4 compute-hour + 100 GB disk]

Preparation: Follow the steps in Section A.3.1 to set up the environment, including downloading the APKs.

Execution: Run `reproducibility/e4.sh <APK_DIR> <OUT_DIR>`, where `<APK_DIR>` is the directory containing the APKs and `<OUT_DIR>` is the output directory for results. The script executes the vulnerable app detection pipeline on the app dataset.

Results: The script should output `OK`.

Alternatively, manually verify that 76.3% ±10% of analyzed apps are vulnerable by inspecting the `vulntapAmountAppsMinOneActivity VulnerablePercent` macro in the generated report located at (`<OUT_DIR>/report.tex`). The error margin accounts for possible variability introduced by analyzing only a subset of apps.

A.5 Notes on Reusability

To foster future research and make it easier for others to build on our work, we provide detailed documentation in the `README.md` files included in each subdirectory. These files include troubleshooting information, describe how to adjust the analysis pipeline (e.g., changing the level of parallelism), explain the code organization and module structure, and outline usage outside Docker environments.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Distributional Private Information Retrieval

Ryan Lehmkuhl
MIT

Alexandra Henzinger
MIT

Henry Corrigan-Gibbs
MIT

A Artifact Appendix

A.1 Abstract

A private-information-retrieval (PIR) scheme lets a client fetch a record from a remote database without revealing which record it fetched. Classic PIR schemes treat all database records the same but, in practice, some database records are much more popular (i.e., commonly fetched) than others. We introduce distributional PIR, a new type of PIR that can run faster than classic PIR—both asymptotically and concretely—when the popularity distribution is heavily skewed. Distributional PIR provides exactly the same cryptographic privacy as classic PIR. The speedup comes from a relaxed form of correctness: distributional PIR guarantees that in-distribution queries succeed with good probability, while out-of-distribution queries succeed with lower probability.

We construct a distributional-PIR scheme that makes black-box use of classic PIR protocols. On a popularity distribution built from real-world data, we show that distributional PIR reduces compute costs by 5–77× compared to existing techniques. Additionally, we present new optimizations to the state-of-the-art classical PIR protocol that decreases preprocessing and query generation time by by 128–351×. Finally, we build CrowdSurf, an end-to-end system for privately fetching tweets, and show that distributional-PIR reduces the end-to-end server cost by 8×.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

N/A

A.2.2 How to access

Our artifact is available at:

- https://github.com/ryanleh/crowdsurf, and
- https://zenodo.org/records/14642111.

For the sake of the artifact evaluation, please use the GitHub link as we'll be pushing any updates, bugfixes, etc. there.

A.2.3 Hardware dependencies

To replicate the exact numbers in the paper, you will need to access to the following AWS machines:

- a `c7i.2xlarge` instances,
- a `c7i.4xlarge` instance,
- a `r7i.4xlarge` instance, and
- a `p3.2xlarge` instance.

If you are unable to access these instances on your own, we are happy to coordinate running these machines for you, just let us know on HotCRP.

Note that running our code does not require access to these exact machines: in each experiment we describe the minimal hardware constraints necessary if you want to try and run on different machines.

A.2.4 Software dependencies

Please see the artifact's README.md.

A.2.5 Benchmarks

None.

(Two of our main results rely on a real-world popularity distribution, however, we don't have permission to publicly release these. Instead, in the artifact we have hardcoded the relevant parameters of our scheme when run on these distributions.)

A.3 Set-up

All machines used should accept TCP traffic on ports 8728 and 8729.

A.3.1 Installation

Please see the artifact's README.md.

A.3.2 Basic Test

Please see the artifact's README.md.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our PIR optimizations reduce SimplePIR's preprocessing time by 116–351× and query generation time by 128–349×. This is proven by the experiment (E1) described in Section 7.1 and whose results are displayed in Figures 3 and 4.

(C2): On the real-world Twitter popularity distribution, our distributional PIR scheme reduces computational costs by 5–77× and communication costs by 8.1–95× compared to batch codes when instantiated with SimplePIR. This is proven by the experiment (E2) described in Section 7.2.1 whose results are displayed in Figure 9.

(C3): On the real-world Twitter popularity distribution, our system, CrowdSurf, allows users to privately fetch tweets for 8× less than existing techniques. This is proven by experiment (E3) described in Section 9.1 and whose results are displayed in Table 12.

A.4.2 Experiments

(E1): *LHE Microbenchmarks [10 human-minutes + 30 compute-minutes]*: This experiment should be run on a `c7i.4xlarge` AWS instance, or any Intel-based machine that supports AVX512 intrinsics with ∼ 32GB of memory.
How to: See the artifact's README.md.
Results: The reported improvement for our scheme should match the claims in C1.

(E2): *Distributional PIR Microbenchmarks [10 human-minutes + 3 compute-hours]*: This experiment should be run on a `r7i.4xlarge` AWS instance, or any instance with ∼ 128GB of memory. *(Note that one can significantly reduce the running time of this experiment by omitting the numbers for the Cuckoo-based batch code as outlined in the artifact README.)*
How to: See the artifact's README.md.
Results: The outputted numbers should visually match Figure 9.

(E3): *CrowdSurf Evaluation [1 human-hour + 1 compute-hour]*: This experiment benchmarks the costs of running a single shard of the full CrowdSurf system–the total end-to-end costs are interpolated from these costs. Recreating the numbers from the paper requires two `c7i.2xlarge` instance, and a `p3.2xlarge` instance. If desired, one could accurately estimate these results without the GPU machine (described in the abstract's README.md).
How to: See the artifact's README.md.
Preparation: All three machines will need to be launched simultaneously, allow TCP traffic on ports 8728 and 8729, and have the correct IP addresses set in the config file described in the abstract's README.md.
Results: The produced numbers should match the numbers from Table 12.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Practical Keyword Private Information Retrieval from Key-to-Index Mappings

Meng Hao[1], Weiran Liu[2], Liqiang Peng[2], Cong Zhang[3] (✉), Pengfei Wu[1], Lei Zhang[2], Hongwei Li[4], and Robert H. Deng[1]

[1]School of Computing & Information Systems, Singapore Management University
[2]Alibaba Group
[3]Institute for Advanced Study, BNRist, Tsinghua University
[4]Peng Cheng Laboratory

A Artifact Appendix

A.1 Abstract

In this artifact appendix, we introduce the minimal hardware and software requirements for the artifact. Since the artifact related to our paper is written in pure Java programming language, the implementation is cross-platform and evaluators can try the artifact on any platform(s) with JDK 17 (or later) installed. We also introduce the source code structure and the ways to run unit tests. We describe how to reproduce the performance reports shown in our paper. Large-scale experiments require relatively large RAM, but evaluators can still try small-scale experiments even on laptops. We welcome suggestions and performance reports on other platforms with future reproducibility.

A.2 Description & Requirements

The artifact contain the implementations of our three constructions: KPIR^{kvs}, $\text{KPIR}^{\text{hash}}$, and $\text{KPIR}^{\text{index}}$, together with the baseline construction ChalametPIR (CCS'24). Evaluators can use the artifact to reproduce Table 1 to Table 6 shown in our paper. We note that evaluators cannot reproduce column "**Memory**" shown in Table 2. The reason is that the tool for reporting the memory usage is jol[1], whose license is not compatible with our license. If evaluators are familiar with Java, it would be relatively easy to manually include related dependencies, add required source codes, and obtain the memory usage report based on the artifact.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact of our paper has been open-sourced and merged into mpc4j (version 1.1.3)[2]. The stable Zenodo URL for this artifcate is available at https://zenodo.org/records/14722434. Evaluators can either visit the stable Zenodo URL or the stable version under tag v1.1.3 to reproduce the experiment results shown in the paper.

Package "edu.alibaba.mpc4j.s2pc.pir.cppir.ks.simple" of the submodule mpc4j-s2pc-pir contains the implementations of our three constructions: KPIR^{kvs} (source codes start with SimpleNaive), $\text{KPIR}^{\text{hash}}$ (source codes start with SimpleBin), $\text{KPIR}^{\text{index}}$ (source codes start with SimplePgm). Package "edu.alibaba.mpc4j.s2pc.pir.cppir.ks.chamelet" of the same submodule contains the implementation of the baseline construction ChalametPIR (CCS'24). Evaluators can also find unit tests for all client-preprocessing keyword PIR protocols in the test folder of package "edu.alibaba.mpc4j.s2pc.pir.cppir.ks". Evaluators can find an example configuration file in the resource folder, which is used for running and reproducing performance reports.

A.2.3 Hardware dependencies

Since implementations related to our paper is written in pure Java programming language, the artifact is expected to support any platforms with JDK 17 (or later) installed. We have tested functionalities of the artifact on 64-bit macOS, Ubuntu, and CentOS systems.

We run our experiments on two machines, each equipped with an Intel Core i9-9900K processor running at 3.6GHz and 128GB of memory. Evaluators may encounter java.lang.OutOfMemoryError when running artifacts on small RAM machines under large parameters.

✉Corresponding author

[1]https://openjdk.org/projects/code-tools/jol/

[2]https://github.com/alibaba-edu/mpc4j/

A.2.4 Software dependencies

The only software dependency is JDK 17 (more later). We note that unit tests also cover other keyword PIR protocol functionalities in the artifact. Therefore, evaluators may encounter errors when running unit tests for protocols that are not related to our paper. Nevertheless, evaluators can still try other protocol implementations after following the guidelines described in the readme to install and configure required native libraries.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The artifact is implemented in a way that chooses the most efficient protocol parameters. Unfortunately, this means that reproducing some performance reports shown in our paper requires modifying source codes. Therefore, we highly recommend loading the artifact using IntelliJ IDEA, the leading Java IDE, so that one can later modify source codes more easily. Evaluators can follow Section **Development** shown in the readme to configure, compile, and run unit tests in IntelliJ IDEA. Evaluators who are familiar with Java can alternatively download and compile the artifact with the following steps.

1. Clone the repository: `git clone https://github.com/alibaba-edu/mpc4j.git`.
2. Go to the root path: `cd mpc4j`.
3. Package the code to get the jar file: `mvn package`.

A.3.2 Basic Test

In package "edu.alibaba.mpc4j.s2pc.pir.cppir.ks", evaluators can find two unit tests, simple/SimpleCpKsPirParamsTest.java, CpKsPirTest.java, in the test folder. Directly pressing the green arrows showing on the left of these source codes can run these unit tests.

Unit tests in simple/SimpleCpKsPirParamsTest.java are used to reproduce columns "**# Row**" and "**# Column**" shown in Table 3, Table 4, and Table 5, while the column "Exp. Rate" can be manually computed based on the report. Since we implement the matrix operations in the transposed form (due to efficiency consideration), we reversely reported "**# Row**" and "**# Column**" in our paper.

Unit tests in CpKsPirTest.java are used to test the functionality of the artifact. Directly run them in IntelliJ IDEA and see if all tests for KPIR$^{\text{kvs}}$ (SIMPLE_NATIVE), KPIR$^{\text{hash}}$ (SIMPLE_BIN), KPIR$^{\text{index}}$ (PGM_INDEX) can pass.

A.4 Evaluation workflow

A.4.1 Major Claims

To reproduce the performance results shown in our paper, evaluators need to generate the jar file (with the name mpc4j-s2pc-pir-1.1.3-jar-with-dependencies.jar) by executing `mvn package`, and run it with two progresses on a single machine, or two progresses on two machines connected by the network. Evaluators also need to create a config file with suitable parameters. Below is a template configuration file.

```
# server information
server_name = server
server_ip = 192.168.1.1
server_port = 9002

# client information
client_name = client
client_ip = 192.168.1.2
client_port = 9003

# append string in the output file
append_string = example

# protocol type
pto_type = SINGLE_CP_KS_PIR

# protocol config
entry_bit_length = 256
server_log_set_size = 22,22,20,18
query_num = 100
parallel = false

# protocol name
single_cp_ks_pir_pto_name = PGM_INDEX
# or CHALAMET, SIMPLE_NAIVE, SIMPLE_BIN
```

(C1): Unit tests in simple/SimpleCpKsPirParamsTest.java can reproduce Table 3, Table 4, and Table 5.

(C2): Running jars with suitable parameters can reproduce Table 1.

(C3): Running jars with suitable source code modifications can reproduce Table 2 and Table 6 (except column "**Memory**", see Section A.2).

A.4.2 Experiments

(E1): [10 human-minutes + 0.1 compute-hours]: Reproduce Table 3, Table 4 and Table 5.
How to: See Section A.3.2.
Results: The numbers shown in the console should be the same as ones listed in Table 3, Table 4, and Table 5.

(E2): [15 human-minutes + 3 compute-hour]: Reproduce Table 1.

How to: Run two processes on two machines connected via network cards, forming a realistic LAN network with a bandwidth of 2.5Gbps and an RTT latency of 0.4ms.

Preparation: Setup two machines. Create a config file shown in Section A.4.1 with correct server/client information, required entry bit length, set sizes, query nums, and protocol names. We assume the config file name is YOUR_CONFIG_FILE_NAME.conf. Place the jar file and the config file under the same path.

Execution: Open one terminal on each of the machines, one for the server and one for the client. Switch to the dictionary where mpc4j-s2pc-pir-1.1.3-jar-with-dependencies.jar and YOUR_CONFIG_FILE_NAME.conf are located. For the server's terminal, execute java -jar mpc4j-s2pc-pir-1.1.3-jar-with-dependencies.jar config_file_name.conf server. For the client's terminal, execute java -jar mpc4j-s2pc-pir-1.1.3-jar-with-dependencies.jar YOUR_CONFIG_FILE_NAME.conf client. Wait until the processes finish. The performance reports are located in the folder "temp".

Results: The performance reports should be similar with ones listed in Table 1.

(E3): [30 human-minutes + 6 compute-hours]: Reproduce Table 2 and Table 6.

How to: Same as E3.

Preparation: To reproduce Table 2, evaluators need to manually assign the value of ε range to 4, 8, 16, 32 sequentially by changing line 30 of the code simple/SimplePgmCpKsPirDesc.java with suitable ε range and re-compile the jar file. To reproduce Table 6, evaluators need to manually assign the dimension of LWE as 1408 by changing line 44 of the code simple/SimpleNaiveCpKsPirConfig.java, simple/SimpleBinCpKsPirConfig.java, and SimplePgmCpKsPirConfig.java from `this(GaussianLweParam.N_1024_SIGMA_6_4);` to `this(GaussianLweParam.N_1408_SIGMA_6_4);` and re-compile the jar file. The remains are same as E3.

Execution: Same as E3.

Results: The performance reports should be similar with ones listed in Table 2 and Table 6.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX'25 Artifact Appendix: Fast Enhanced Private Set Union in the Balanced and Unbalanced Scenarios

Binbin Tu[1,2,3], Yujie Bai[1,2,3], Cong Zhang[4], Yang Cao[1,2,3], and Yu Chen[1,2,3](✉)

[1]School of Cyber Science and Technology, Shandong University, Qingdao 266237, China
[2]Quan Cheng Laboratory, Jinan 250103, China
[3]Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao 266237, China
[4]Institute for Advanced Study, BNRist, Tsinghua University, Beijing, China
{tubinbin,yuchen}@sdu.edu.cn, {baiyujie,caoyang24}@mail.sdu.edu.cn, zhangcong@mail.tsinghua.edu.cn

A Artifact Appendix

A.1 Abstract

The source code for our two enhanced PSU (ePSU) protocols in balanced and unbalanced settings, is available at https://github.com/real-world-cryptography/ePSU. We implement the hashing, pnMCRG, and one-time pad routines for our ePSU protocols. Our pnMCRG consists of two steps: pMCRG and nECRG. Additionally, pMCRG can be further divided into three steps: bOPRF + OKVS + pECRG in the balanced setting and bOPRF + FHE + pECRG in the unbalanced setting. All artifact evaluation results are provided in Section 8 of the paper and the Appendix.

A.2 Description & Requirements

The source code for our balanced and unbalanced ePSU protocols can be described as follows.

Balanced Setting. The code of our balanced ePSU is located in the "balanced_ePSU/" folder. In this folder, "ePSU/balanced_epsu.cpp" contains the complete implementation of ePSU, and "pnMCRG/pnMCRG.cpp" implements pnMCRG (Figure 18: pnMCRG from pMCRG and nECRG), including key components pMCRG (Figure 13: pMCRG from bOPRF, OKVS, and pECRG), pECRG (Figure 11: DDH-based pECRG), and nECRG (Figure 16: nECRG from ssPEQT and ROT). The "test/" folder contains test programs for balanced ePSU, pnMCRG, pMCRG, pECRG, and nECRG.

Unbalanced Setting. The code of our unbalanced ePSU is located in the "unbalanced_ePSU/" folder. In the "MCRG/" folder, we reuse part of APSU (https://github.com/real-world-cryptography/APSU) to implement steps 1–7 from Figure 14 (MCRG based on bOPRF and FHE). "pECRG_nECRG_OTP/pecrg_necrg_otp/pecrg_necrg_otp.cpp" implements the remaining parts of our unbalanced ePSU. "pECRG_nECRG_OTP/pnecrg/pnECRG.cpp" implements the key components pECRG and nECRG. The "test/" folder contains test programs for pECRG (step 8 in Figure 14) and nECRG (Figure 16: nECRG from ssPEQT and ROT). Additionally, "test.py" is a script to automate the MCRG and pECRG_nECRG_OTP processes.

We run our experiments on a single Intel Core i7-13700 CPU @ 5.20GHz with 32 threads and 64GB of RAM, running Ubuntu 22.04.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

We have open-sourced the implementation of our two enhanced PSU protocols at https://github.com/real-world-cryptography/ePSU. Besides, our artifact is available at https://doi.org/10.5281/zenodo.15128481.

A.2.3 Hardware dependencies

Our ePSU currently supports Ubuntu 22.04.

A.2.4 Software dependencies

Our ePSU requires the installation of G++ (version 11.4.0) and Python (version 3.10.12).

We leverage the constructions in [KKRT16, RS21, RR22, FV12] to implement bOPRF, OKVS, ssPEQT, and FHE (the building block of balanced/unbalanced pnMCRG) and use the code from the Vole-PSI library [Vol], the SEAL library [SEA], and the APSI/APSU library [APS, PSU]. We use an array of optimization techniques of FHE as [CLR17, CHLR18, CMdG+21, TCLZ23], such as batching (SIMD), windowing, and partitioning to significantly reduce the depth of the homomorphic circuit. As for ROT (the building block of nECRG) and DDH-based pECRG, we follow the libOTe library [lib]

and the OpenSSL library [Ope]. Our code supports multi-threading parallelism following the Vole-PSI library [Vol] and the OpenMP library [MP].

A.2.5 Benchmarks

We set the computational security parameter $\kappa = 128$, the statistical security parameter $\lambda = 40$ and use $\gamma = 3$ hash functions to insert sets X and Y into the cuckoo hash table and simple hash table, respectively. The item length is 64-bit following [LG23, DCZB24].

A.3 Set-up

A.3.1 Installation

Instructions for installing the required dependencies are given in https://github.com/real-world-cryptography/ePSU/blob/main/README.md. This code is conducted on Ubuntu 22.04, with G++ 11.4.0 and CMake 3.22.1.

A.3.2 Basic Test

ePSU Test. To run our balanced ePSU, evaluators can run the command in the balanced_ePSU/build directory. For example, when both set sizes are $|X| = |Y| = 2^{12}$ and thread number is 1, the command is `./test_balanced_epsu -nn 12 -nt 1 -r 0 & ./test_balanced_epsu -nn 12 -nt 1 -r 1`, where `nn`$\in \{10, 12, 14, 16, 18, 20, 22\}$ denotes the logarithm of set size $\{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}\}$; `nt` denotes the number of threads$\{1, 2, 4, 8\}$; `r` denotes the index of involved parties (only 0 or 1).

Sub-protocols of ePSU test: run the commands in the balanced_ePSU/build directory. The symbols `nn`, `nt`, and `r` remain consistent with those in ePSU.

- **pnMCRG Test Command.**
  ```
  ./test_pnmcrg -nn 12 -nt 1 -r 0 &
  ./test_pnmcrg -nn 12 -nt 1 -r 1
  ```
- **pMCRG Test Command.**
  ```
  ./test_pmcrg -nn 12 -nt 1 -r 0 &
  ./test_pmcrg -nn 12 -nt 1 -r 1
  ```
- **pECRG Test Command.**
  ```
  ./test_pecrg -nn 12 -nt 1 -r 0 &
  ./test_pecrg -nn 12 -nt 1 -r 1
  ```
- **nECRG Test Command.**
  ```
  ./test_necrg -nn 12 -nt 1 -r 0 &
  ./test_necrg -nn 12 -nt 1 -r 1
  ```

eUPSU Test. To run our unbalanced ePSU, evaluators can run the command in the ePSU/unbalanced_ePSU directory. Here we fix the small set size $|X| = 2^{10}$. For example, when the large set size $|Y| = 2^{12}$ and thread number is 1, the command is `python3 test.py -pecrg_necrg_otp -cn 1 -nt 1 -nn 12`, where `nn`$\in \{12, 14, 16, 18, 20, 22\}$ denotes the logarithm of large set size $\{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}\}$; `nt` denotes the number of threads $\{1, 2, 4, 8\}$; `cn` denotes the column number of the matrix (if the large set size is less than 2^{20}, set to 1; otherwise set to 2).

Sub-protocols of eUPSU test: run the commands in the ePSU/unbalanced_ePSU directory. The symbols are the same as above.

- **pECRG Test Command.**
  ```
  python3 test.py -pecrg -cn 1 -nt 1 -nn 12
  ```
- **pnECRG Test Command.**
  ```
  python3 test.py -pnecrg -cn 1 -nt 1 -nn 12
  ```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We implement our enhanced PSU in the balanced setting (ePSU), and its subprotocols, including pnMCRG, pMCRG, pECRG, and nECRG.

(C2): We implement our enhanced PSU in the unbalanced setting (eUPSU), and its subprotocols, including pnMCRG and pMCRG.

A.4.2 Experiments

(E1): *[ePSU performance] : This experiment benchmarks the communication and runtime of ePSU with 10Gbps bandwidth, 0.4ms RTT; 100Mbps, 10Mbps and 1Mbps bandwidth, 160ms RTT.*

How to: Open a terminal, and execute the following command: `tc qdisc add dev lo root netem delay 0.2ms rate 10Gbit` to config the local network as 10Gbit bandwidth with 0.4ms RTT. Evaluators can adjust the network settings using different parameters as needed, such as 100Mbps, 10Mbps and 1Mbps bandwidth, 160ms RTT. Then run the command `./test_balanced_epsu -nn 20 -nt 1 -r 0 & ./test_balanced_epsu -nn 20 -nt 1 -r 1` in the balanced_ePSU/build directory. The command will run balanced_ePSU with set size $|X| = |Y| = 2^{20}$, and print the information of communication and running time. Evaluators can set `nn` to test balanced_ePSU under different set sizes, `nt` to test balanced_ePSU with different numbers of threads.

Results: For each run of balanced_ePSU, this experiment print information as follows:

- `Comm cost = 277.402 MB`, indicating the communication cost of the balanced_ePSU is 277.402 MB.
- `end 78144.0 78143.895 **********`, indicating the running time of the balanced_ePSU is 78144 ms.

- `Balanced_ePSU functionality test pass! And union size is: 1048577`, indicating the Balanced_ePSU functionality tests passed.

(E2): *[eUPSU performance]: This experiment benchmarks the communication and runtime of eUPSU with 10Gbps bandwidth, 0.4ms RTT; 100Mbps, 10Mbps and 1Mbps bandwidth, 160ms RTT.*

How to: Open a terminal, and execute the following command: `tc qdisc add dev lo root netem delay 0.2ms rate 10Gbit` to config the local network as 10Gbit bandwidth with 0.4ms RTT. Evaluators can adjust the network settings using different parameters as needed, such as 100Mbps, 10Mbps and 1Mbps bandwidth, 160ms RTT. Then run the command `python3 test.py -pecrg_necrg_otp -cn 1 -nt 1 -nn 20` in the unbalanced_ePSU directory. The command will run unbalanced_ePSU with set size $|X| = 2^{10}, |Y| = 2^{20}$, and print the information of communication and time. Evaluators can set `nn` to test unbalanced_ePSU under different set sizes, `nt` to test unbalanced_ePSU with different numbers of threads.

Note: The total running time of unbalanced_ePSU is the sum of the MCRG execution time and the pECRG_nECRG_OTP execution time. Similarly, the total communication cost of unbalanced_ePSU is the sum of the MCRG communication cost and the pECRG_nECRG_OTP communication cost.

Results: For each run of unbalance_ePSU, this experiment print information as follows:

- `receiver all time15778.6`, indicating the running time of MCRG in unbalanced_ePSU is 15778.6 ms.
- `end 178.4 178.382 **********`, indicating the running time of pECRG_nECRG_OTP in unbalanced_ePSU is 178.5 ms.
- `Communication total: 1983 KB`, indicating the communication cost of MCRG in unbalanced_ePSU is 1983 KB.
- `Comm cost = 0.482 MB`, indicating the communication cost of pECRG_nECRG_OTP in unbalanced_ePSU is 0.482 MB.

(E3): *[balanced pnMCRG performance]: This experiment benchmarks the communication and runtime of pnMCRG in a balanced setting with 10Gbps bandwidth, 0.4ms RTT.*

How to: Open a terminal, and execute the following command: `tc qdisc add dev lo root netem delay 0.2ms rate 10Gbit` to config the local network as 10Gbit bandwidth with 0.4ms RTT. Then run the command `./test_pnmcrg -nn 20 -nt 1 -r 0 & ./test_pnmcrg -nn 20 -nt 1 -r 1` in the balanced_ePSU/build directory. The command will run the balanced pnMCRG with set size $|X| = |Y| = 2^{20}$, and print the information of communication and time. Evaluators can set `nn` to test the balanced pnMCRG under different set sizes, `nt` to test the balanced pnMCRG with different numbers of threads.

Results: For each run of balanced pnMCRG, this experiment prints information as follows:

- `Comm cost = 256.355 MB`, indicating the communication cost of the balanced pnMCRG is 256.355 MB.
- `end 78065.2 78065.166 **********`, indicating the running time of the balanced pnMCRG is 78065.2 ms.
- `pnMCRG functionality test pass!`, indicating the balanced pnMCRG functionality tests passed.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[APS] https://github.com/microsoft/APSI.

[CHLR18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM CCS*, 2018.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *ACM CCS*, 2017.

[CMdG+21] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In *ACM CCS*, 2021.

[DCZB24] Minglang Dong, Yu Chen, Cong Zhang, and Yujie Bai. Breaking free: Efficient multi-party private set union without non-collusion assumptions. *IACR Cryptol. ePrint Arch.*, 2024.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *ACM CCS*, 2016.

[LG23] Xiang Liu and Ying Gao. Scalable multi-party private set union from multi-query secret-shared private membership test. In *ASIACRYPT*, 2023.

[lib] https://github.com/osu-crypto/libOTe.

[MP] https://www.openmp.org/.

[Ope] https://github.com/openssl/openssl.

[PSU] https://github.com/real-world-cryprography/APSU.

[RR22] Srinivasan Raghuraman and Peter Rindal. Blazing fast PSI from improved OKVS and subfield VOLE. In *ACM CCS*, 2022.

[RS21] Peter Rindal and Phillipp Schoppmann. VOLE-PSI: fast OPRF and circuit-psi from vector-ole. In *EUROCRYPT*, 2021.

[SEA] https://github.com/microsoft/SEAL.

[TCLZ23] Binbin Tu, Yu Chen, Qi Liu, and Cong Zhang. Fast unbalanced private set union from fully homomorphic encryption. In *ACM CCS*, 2023.

[Vol] https://github.com/Visa-Research/volepsi.



USENIX Security '25 Artifact Appendix: Your Shield is My Sword: A Persistent Denial-of-Service Attack via the Reuse of Unvalidated Caches in DNSSEC Validation

Shuhan Zhang*‡, Shuai Wang†, Li Chen†, Dan Li*, Baojun Liu*
*Tsinghua University, †Zhongguancun Laboratory
‡Tsinghua Shenzhen International Graduate School

A Artifact Appendix

A.1 Abstract

This artifact is the Proof-of-Concept of the RUC attack. It includes the Dockerfiles of the tested DNS software and the victim domains' nameserver, scripts for the three RUC attack variants, and detailed instructions to facilitate the vulnerability reproduction within a controlled environment.

A.2 Description & Requirements

In this section, we first discuss the security, privacy and ethical concerns of our artifact. Then, we provide its access link, and demonstrate its hardware and software dependencies.

Specifically, our artifact consists of the following components:

- Dockerfiles of the tested DNS software and authoritative nameservers of the victim domains (see "dockers/").
- Scripts to launch each of the three RUC attack variants within the controlled environment (see "poc_scripts/").
- Detailed instructions on the setup of the test environment (see "environment_setup.md") and steps to reproduce three RUC variants (see "ruc_reproduction.md").

Details about the artifact composition and the file structure are demonstrated in the file "README.md".

A.2.1 Security, privacy, and ethical concerns

First, for the tested DNS resolver software, we collect the publicly available version of each software from the vendor's official website. All the tests are performed within a controlled environment set up on the local machine. Specifically, for open-source software, we build Docker images and run them in containers. For closed-source software, *e.g.*, Microsoft DNS, we provide a virtual machine with the DNS service installed. Second, for the victims of the RUC attack, the tests in our artifact target the victim domains under our controlled apex zone. We simulate an on-path attacker located with the victim domains' nameserver in a local Docker container, ensuring no harm to DNS domains in the real world. Third, we have excluded any scripts or datasets that could potentially be exploited by adversaries to launch real-world attacks, *e.g.*, the scripts for large-scale open resolver measurement. Moreover, this artifact does not involve any sensitive data such as personal information. Therefore, we believe that this artifact does not raise ethical issues.

A.2.2 How to access

We have released our artifact on Zenodo, which can be accessed at https://doi.org/10.5281/zenodo.15543846. Please refer to the latest version of our artifact as automatically directed by this concept DOI.

A.2.3 Hardware dependencies

This artifact can run in standard CPU-based environments, and no specialized hardware is required. Nevertheless, to ensure the successful build of the DNS software docker images, we recommend that the host machine for setting up the Linux-based test environment is equipped with a number of CPU cores ≥ 4 and RAM $\geq$ 8GB.

For Microsoft DNS, we provide a virtual machine .ova file that can be imported into a virtual machine tool, *e.g.*, VMware Workstation. Note that this virtual machine is only compatible with host machines based on x86 architecture.

A.2.4 Software dependencies

All the Dockerfiles in this artifact are built on Ubuntu 22.04 with Docker Engine. The Docker Engine can be installed on the host machine using the following command:

```
apt install docker.io
```

The scripts for testing RUC vulnerabilities are executed in Python environment, *e.g.*, Python 3.10.12. All the required Python packages are listed in "poc_scripts/requirement.txt", which can be installed using the `pip` command.

A.2.5 Benchmarks

None.

A.3 Set-up

This section describes the environment setup for the RUC vulnerability reproduction. For detailed, specific explanations of the set-up steps, please refer to "environment_setup.md".

A.3.1 Installation

Resolvers running on Linux. A Docker environment is set up to test the five Linux-based DNS resolvers.

First, create a Docker network named `ruc-test-net` on the local host machine:

```
docker network create --subnet "172.22.0.0/16"
                              ruc-test-net
```

This network is dedicated to the RUC vulnerability test. All DNS components, *i.e.*, the tested resolvers, the authoritative nameserver of the victim domains and the client, will be assigned with an IP address in the subnet `172.22.0.0/16`.

Next, under the base folder, run the following commands to build Docker images and run containers for all the tested Linux-based resolvers:

```
bash start_resolver.sh
```

Microsoft DNS resolver. For Microsoft DNS, we offer a ready-made Windows Server virtual machine (.ova file[1]) with Microsoft DNS installed and DNSSEC validation enabled. After importing it into a virtual machine tool such as VMware Workstation, you can log in to the server as `Administrator` with password `RUC@Sec25`. We have put the test scripts on the Desktop of the virtual machine, *i.e.*, under "C:\Users\Administrator\Desktop\poc_scripts\". All necessary python packages have been installed in advance.

If you intend to manually install Microsoft DNS on a Windows Server virtual machine and configure DNSSEC validation, please refer to the detailed instructions in the manual "dockers/resolver_software/microsoft/microsoft_setup.pdf".

Victim domains and nameserver. For Linux-based resolvers, we support the vulnerability test using our controlled apex zone, `dnssec-ruc.xyz`. Each RUC attack variant corresponds to a dedicated victim subdomain under `dnssec-ruc.xyz`. The authoritative nameserver of these subdomains will be automatically resolved to `172.22.2.1` as configured at our apex zone, *i.e.*, the IP address of the local nameserver container.

Run the following command to build the Docker image for the local nameserver and start a container:

```
bash start_nameserver.sh
```

[1] https://doi.org/10.5281/zenodo.15509714

The container runs the latest version of BIND to provide authoritative DNS service. It also runs an API for the RUC attack, which is invoked by the attack script to simulate on-path response manipulations. Notably, the RUC vulnerabilities of DNS resolvers are independent of specific victim domains, and all resource records of the victim domains under our apex zone are transparent. Please refer to the apex zone file "victim_config/apex_zone.txt" for their referral delegation records (*e.g.*, `NS`, `DS` records) and zone files under "victim_config/bind/" for their authoritative records.

For Microsoft DNS, the victim subdomains are under our controlled zone `dnssec-ruc-ms.xyz`. Their nameserver will be automatically resolved to `47.251.171.85`, *i.e.*, our own DNS server dedicated to testing Microsoft DNS.

RUC attacker. Use the following command to build a Docker image and start a container for the RUC attacker:

```
bash start_attacker.sh
```

This command will mount the PoC scripts ("poc_scripts/") under the "/root/" directory in the container. For simplicity, we simulate DNS queries of ordinary clients to verify the RUC attack within the same container as the attacker.

Pull ready-made docker images from DockerHub. Alternatively, you can pull the ready-made Docker images of the resolver software (except Microsoft DNS), nameserver and RUC attacker from DockerHub (see "dockers/dockerhub.md").

A.3.2 Basic Test

To verify the environment setup, we issue DNS queries from the RUC attacker container to each tested resolver, requesting a DNSSEC-signed domain hosted on the local nameserver. Run the following commands for this basic test:

```
cd poc_scripts
bash test_basic.sh
```

The outputs of the basic test will be stored in "poc_scripts/basic_test_result/log_basic_test.csv". All entries should be tagged as "succeed" if the setup is correct, *i.e.*, the tested resolvers are running with DNSSEC validation enabled, and the nameserver is also functioning properly.

For Microsoft DNS, on the Windows Server virtual machine, enter the folder "poc_scripts" located on the Desktop and run the script "test_basic_microsoft.ps1" using PowerShell. The outputs will be stored in "log_basic_test-microsoft.csv" under the folder "basic_test_result". Similarly, there should be "succeed" in the output, indicating a successful environment setup.

A.4 Evaluation workflow

This section illustrates the steps to reproduce the three RUC attack variants (see more details in "ruc_reproduction.md").

A.4.1 Major Claims

(C1): The vulnerabilities of the tested DNS resolver software against each RUC variant are consistent with those detailed in Section 5.1, Table 2 of the main paper. For Linux-based resolvers, the reproduction of attack variants $\mathsf{RUC}_{\mathsf{SEC}}$ ($\mathsf{RUC}_{\mathsf{DNSKEY}}$ and $\mathsf{RUC}_{\mathsf{DS}}$), $\mathsf{RUC}_{\mathsf{NSIP}}$, and $\mathsf{RUC}_{\mathsf{EDNS0}}$ corresponds to experiment **E1**, **E2**, **E3** and **E4**, respectively. For Microsoft DNS, the reproduction of all attack variants corresponds to experiment **E5**.

A.4.2 Experiments

The PoC script of the RUC attack is "poc_scripts/ruc_poc.py". The workflow of this script is as follows:

1. `config_authns('inject')`: A response manipulation program is started by invoking an API (see "victim_config/attack_api/") on the victim domain's nameserver, simulating a (short-lived) on-path attacker.
2. `inject_cache()`: The attacker injects forged, unvalidated data into the cache of the target resolver, *e.g.*, `DNSKEY` without `RRSIG` for an $\mathsf{RUC}_{\mathsf{DNSKEY}}$ attack.
3. `config_authns('resume')`: The manipulation program at the nameserver side is stopped, and the original, benign service of the nameserver is resumed.
4. `verify_dos()`: The ordinary client sends routine DNS queries demanding DNSSEC validation to the target resolver. Responses from the resolver are recorded and checked if they carry valid answers. If no answer is returned (*i.e.*, `SERVFAIL` or timeout), the resolver is vulnerable to the corresponding RUC variant.

The test results of the RUC vulnerabilities are logged in "poc_scripts/ruc_test_result/log_ruc_test.csv" (for Microsoft DNS, the file name is "log_ruc_test-microsoft.csv"). Each line of the file is in the following format:

```
{resolver software}|{resolver IP}|{RUC variant}|
{with RRSIG}|{vulnerable}|{query failure rate}
```

The field "**vulnerable**" indicates the vulnerability of the tested resolver against the specific RUC attack variant, which should be consistent with Table 2 of the main paper.

For testing Linux-based resolvers in **E1-E4**, first enter the folder "poc_scripts":

```
cd poc_scripts
```

Then, run the corresponding test script for each RUC attack variant as follows.

(E1): [Test $\mathsf{RUC}_{\mathsf{DNSKEY}}$ of $\mathsf{RUC}_{\mathsf{SEC}}$] [10 human-minutes]:
Execution:

```
bash test_ruc_dnskey.sh
```

Results: For vulnerable resolvers, the "vulnerable" field in the test result output should be "True", *i.e.*, all responses of the verifying queries should be `SERVFAIL`.

(E2): [Test $\mathsf{RUC}_{\mathsf{DS}}$ of $\mathsf{RUC}_{\mathsf{SEC}}$] [10 human-minutes]:
Execution:

```
bash test_ruc_ds.sh
```

Results: For vulnerable resolvers, the "vulnerable" field in the test result output should be "True", *i.e.*, all responses of the verifying queries should be `SERVFAIL`.

(E3): [Test $\mathsf{RUC}_{\mathsf{NSIP}}$] [5 human-minutes]:
Execution:

```
bash test_ruc_nsip.sh
```

Results: For vulnerable resolvers, the "vulnerable" field in the test result output should be "True", *i.e.*, all responses of the verifying queries should be `SERVFAIL` or time out.

(E4): [Test $\mathsf{RUC}_{\mathsf{EDNS0}}$] [5 human-minutes]:
Execution:

```
bash test_ruc_edns0.sh
```

Results: For vulnerable resolvers, the "vulnerable" field in the test result output should be "True", *i.e.*, all responses of the verifying queries should be `SERVFAIL`.

(E5): [Test Microsoft DNS] [5 human-minutes]:
Execution: Run "C:\Users\Administrator\Desktop\poc_scripts\test_microsoft.ps1" using PowerShell on the Windows Server virtual machine.
Results: For attack variants to which Microsoft DNS is vulnerable resolvers, the "vulnerable" field in the test result output should be "True", *i.e.*, all the response status codes of the verifying queries should be `SERVFAIL`.

If you are interested in a specific resolver, please refer to the commands in "ruc_reproduction.md" to specify the tested resolver and inspect its behavior under a given RUC variant. This document also demonstrates the detailed workflow and the input options of the PoC script.

A.5 Notes on Reusability

Apart from reproducing the proposed RUC vulnerability, we believe that the Dockerfiles and virtual machine of DNS resolvers can also facilitate the testing or reproduction of other DNS vulnerabilities within the controlled environment. Moreover, researchers can reuse the Dockerfiles to install arbitrary versions of the resolver software by simply changing the version number in the corresponding file.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: BGP Vortex: Update Message Floods Can Create Internet Instabilities

Felix Stöger, Henry Birge-Lee, Giacomo Giuliari, Jordi Subira-Nieto, Adrian Perrig

A Artifact Appendix

A.1 Abstract

We provide a comprehensive artifact containing all the resources necessary to reproduce our results from sections 4 and 5 of our work. It is available to download as a ZIP archive via a persistent URL (appendix A.2.2).

A.2 Description & Requirements

The artifact submission contains all scripts and configurations required for evaluation. The ZIP archive contains two folders for replicating results from sections 4 and 5 respectively. Each folder includes a comprehensive README with step-by-step execution instructions and requirements. The artifact provides complete attack simulation components: router configurations, orchestration scripts, and notebooks for data analysis.

Replication of our experiments requires 8 virtual machines. We used DigitalOcean for evaluation. The experiments generate several hundred gigabytes of data, compressed to approximately 50GB. The setup runs on standard VMs using common software packages (e.g., Jupyter Lab), requiring no specialized hardware or software. For the functional and reproduced artifact evaluation, we will provide you with 8 VM instances to ensure the setup is indeed identical to the one used for the experiments. Please contact us, so we can set the VMs up and give you access. To analyze the results, we recommend a machine running Ubuntu 24.04 LTS with a modern CPU, sufficient RAM (16GB minimum) and 100-200GB disk space.

A.2.1 Security, privacy, and ethical concerns

The experiments are self-contained and isolated from any public routing infrastructure. The amount of traffic generated is normal, so our experiments also do not negatively affect the cloud provider.

A.2.2 How to access

You can access the artifacts via the following stable link to Zenodo: https://doi.org/10.5281/zenodo.15612433.

A.2.3 Hardware dependencies

Our experimental setup uses DigitalOcean VMs to eliminate hardware dependencies and ensure reproducibility. For local replication, we recommend matching the following VM specifications:

- **3x CPU-Optimized / 8 GB / 4 vCPUs:** VMs hosting the three ASes not directly part of the BGP Vortex: *bystander-as*, *upstream-as*, and *downstream-as*. Each of these ASes runs the FRR router.
- **4x Memory-Optimized / 16 GB / 2 vCPUs:** VMs hosting the four ASes of the BGP Vortex: *attacker-as*, and its three providers *as-1*, *as-2*, and *as-3*. Each of these ASes runs the BIRD router.
- **1x Storage-Optimized / 16 GB / 2 vCPUs:** VM used for orchestration and data gathering.

The machine on which you intend to evaluate the results of the experiments should be equipped with a modern CPU, sufficient RAM (16GB minimum) and 100-200GB disk space.

A.2.4 Software dependencies

Our experiments run on Ubuntu 24.04 LTS and require BIRD and FRR software routers, iperf3, and tcpdump. Data analysis uses Tshark and Jupyter-Lab, with the following Python packages: `matplotlib`, `mrtparse`, `pandas`, `numpy`, and `seaborn`.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Step 1: Set up local environment Extract the artifact to the machine on which you also intend to perform the analysis of the experiment data. All subsequent directory references are relative to the artifact root unless specified otherwise.

Step 1: Configure VMs. First, provision eight virtual machines with the aforementioned specifications. Ensure co-location within a single data center and, when utilizing cloud infrastructure, verify virtual private cloud (VPC) support. Establish SSH connectivity to all instances and confirm inter-VM communication via private IP addresses.

Afterwards, copy the public and private IP addresses for each VM into `artifact/section_5/generation/ips_map.json` as detailed in the Section 5 README.

Lastly, install FRR, BIRD, and tcpdump on all VMs. On the machine on which you intend to analyze the results, also install tshark and the following Python packages: matplotlib, mrtparse, pandas, numpy, and seaborn.

Step 2: Generate Configuration Files and Scripts. Execute the `artifact/section_5/generation/generate_output.py` script from within the generation folder on your local machine. This populates the output folder with configuration files specific for your deployment.

Step 3: Deploy Configuration to VMs. Execute the `artifact/section_5/generation/push_configs.py` script from within the generation folder on your local machine to deploy the generated configuration files to the VMs.

Step 4: Establish Network Topology. Configure GRE tunnels connecting the upstream-AS, bystander-AS, and downstream-AS. To that end, SSH into the fetcher VM and execute `bash set_up_gretaptunnels.sh`. Subsequently, SSH into the upstream-AS and bind the IP address `192.168.10.1` to its loopback interface. Repeat this step for the downstream-AS with IP `192.168.11.1`.

A.3.2 Basic Test

SSH into the fetcher VM, navigate into the `utils` folder and execute `bash trigger_sending_K_mins.sh 0 0 2 200 8000`. The expected output is:

```
Initializing experiment with 0 prefixes,
200% CPU,and 8000MB memory for 0 minutes.
Setting resource constraints on bystander-as
DEBUG: killing existing instance of
assignResources.sh on the bystander-as
DEBUG: Executing command:
ssh bystander-as "nohup bash ~/assignResources.sh
${cpuCores} ${cpuShare} ${memory} ${swapMemory}
>/dev/null 2>&1 &"
Setting resource constraints on downstream-as
DEBUG: killing existing instance of
assignResources.sh on the bystander-as
DEBUG: Executing command
ssh bystander-as "nohup bash ~/assignResources.sh
${cpuCores} ${cpuShare} ${memory} ${swapMemory}
>/dev/null 2>&1 &"
Setting resource constraints on upstream-as
DEBUG: killing existing instance of
assignResources.sh on the bystander-as
DEBUG: Executing command:
ssh bystander-as "nohup bash ~/assignResources.sh
${cpuCores} ${cpuShare} ${memory} ${swapMemory}
>/dev/null 2>&1 &"
bird_attacker.conf
Started experiment!
Experiment finished, stopping!
DEBUG: killing existing instance of
assignResources.sh on the bystander-as
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): BGP Vortices are widespread on today's Internet (Section 4). We demonstrate this by identifying BGP Vortices in the CAIDA AS-relationship based on known vulnerable ASes.

(C2): BGP Routers are susceptible to BGP Vortices (Section 5.2). We demonstrate the vulnerability of widely deployed, industry-standard router implementations.

(C3): BGP Vortices delay network convergence (Section 5.3). We create a BGP Vortex in our testbed and measure route update propagation delay, demonstrating that critical routing information dissemination is delayed, thus prolonging convergence.

(C4): BGP Vortices overload routers (Section 5.4). We show that our experimental setup, routers fail to keep up with the rate at which route updates would be sent, as evidenced by TCP receive window sizes reaching zero.

(C5): BGP Vortices cause data plane outages (Section 5.5). We create a BGP Vortex in our testbed, measure data plane connectivity using iperf3, simulate route changes, and quantify data plane outage duration.

A.4.2 Experiments

In this section, we describe how to replicate our experiments and how to analyze the results. We encourage the reader to consult the READMEs provided inside the artifact, which include more detailed step-by-step instructions.

(E1): BGP Vortices are widespread on today's Internet (Section 4). [5 human-minutes + 5 compute-seconds]:
Preparation: No preparation required, assuming prior instructions were followed and dependencies installed.
Execution: Execute the following commands from

within the `artifact/section_4/evaluation/` directory:

```
# Count analysis
python run.py -t data/20240301.as-rel.txt
-a data/sico_ases.txt count

# Downstream analysis:
python run.py -t data/20240301.as-rel.txt
-a data/sico_ases.txt downstream
```

Results: The count analysis computes the number of BGP Vortices (gadgets) each of the 21 vulnerable ASes is included in. The downstream analysis computes the number of updates seen by every AS in the customer cone of the 21 vulnerable ASes. Both results together correspond to insights from section 4 of our paper, and thus prove claim C1.

(E2): BGP Routers are susceptible to BGP Vortices (Section 5.2).: We do not provide an explicit experiment, since the subsequent experiments generate BGP Vortices and thus also prove that BGP routers are susceptible.

(E3): BGP Vortices delay network convergence (Section 5.3). [15 human-minutes + > 1 compute-day]

Preparation: No preparation required, assuming prior instructions were followed and dependencies installed.

Execution: To initiate the experiment, SSH into the fetcher VM and execute `bash cp_experiment.sh 2 tcpdump`. The experiment requires substantial time; we recommend execution within a `tmux` instance (on the fetcher VM) to maintain session persistence. Record the experiment identifier as it is needed for subsequent evaluation.

Results: To analyze the experiment, first retrieve data from the fetcher VM by executing the following command from the `artifact/section_5/evaluation/` directory:

```
scp -r root@<fetcher VM IP>:
~/data/dataplane_experiment/<exp. id>
./experiment_data/<ex. id>
```

Each experiment generates a separate file for 100% and 200% CPU share (1 and 2 CPU cores) and for 0, 50, 100, 600, 1000, 3000, and 5000 attacker prefixes. Processing each file sequentially is highly inefficient, so instead we provide a script that should be executed in parallel for each combination of CPU share and attacker prefix count (14 combinations total). Execute the following commands for each combination from the `artifact/output/extraction/` directory:

```
python3 parallel_tcpdump_extraction.py
<exp. id> bystander-as
<cpu share> <num attack pfx> arrival_times

python3 parallel_tcpdump_extraction.py
<exp. id> upstream-as
<cpu share> <num attack pfx> arrival_times
```

To analyze data and generate plots comparable to those presented in the paper, launch a jupyter-lab instance in the `artifact/section_5/evaluation` directory and execute all cells in `section_5_3.ipynb`. Ensure the experiment identifier is correctly configured in the designated notebook cell.

(E4): BGP Vortices overload routers (Section 5.4). [15 human-minutes + several compute-hours]

Preparation: No preparation required, assuming prior instructions were followed and dependencies installed.

Execution: No execution required; this experiment reuses results from experiment E3.

Results: Each experiment contains data for 100% and 200% CPU share (1 and 2 CPU cores) and for 0, 50, 100, 600, 1000, 3000, and 5000 attacker prefixes. Given the computational complexity of processing each file, we provide a script for parallel execution across all combinations of CPU share and attacker prefix counts (14 combinations total). Execute the following command for each combination from the `artifact/output/extraction/` directory:

```
python3 parallel_tcpdump_extraction.py
<exp. id> bystander-as <cpu share>
<num attack pfx> zerowindows
```

To analyze the data and generate plots corresponding to those in the paper, launch a jupyter-lab instance in the `artifact/evaluation` folder and execute all cells in `section_5_4.ipynb`. Verify the experiment identifier is correctly configured in the designated notebook cell.

(E5): BGP Vortices cause data plane outages (Section 5.5). [15 human-minutes + >1 compute-day]

Preparation: No preparation required, assuming prior instructions were followed and dependencies installed.

Execution: To initiate the experiment, SSH into the fetcher VM and execute `bash dp_experiment.sh 2`. The experiment requires substantial time; we recommend execution within a `tmux` instance (on the fetcher VM) to maintain session persistence. Record the experiment identifier as it is needed for subsequent evaluation.

Results: To analyze the experiment, you first have to retrieve the data from the fetcher VM. This is achieved by executing the following command from within the `artifact/section_5/evaluation/` directory:

```
scp -r root@<fetcher VM IP>:
~/data/dataplane_experiment/<exp. id>
./experiment_data/<ex. id>
```

Each experiment holds data for 100% and 200% CPU share (1 and 2 CPU cores), and for 0, 50, 100, 600, 1000, 3000, and 5000 attacker prefixes. Since process-

ing each file takes a long time, we provide a script which we encourage to execute in parallel for each combination of CPU share and number of attack prefixes (14 times in total). Do so by executing the following line for each such combination, from within the `artifact/output/extraction/` directory:
To analyze data and generate plots comparable to those presented in the paper, launch a jupyter-lab instance in the `artifact/section_5/evaluation` directory and execute all cells in `section_5_5.ipynb`. Ensure the experiment identifier is correctly configured in the designated notebook cell.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Are CAPTCHAs Still Bot-hard? Generalized Visual CAPTCHA Solving with Agentic Vision Language Model

Xiwen Teoh[1,2], Yun Lin[1], Siqi Li[2], Ruofan Liu[2], Avi Sollomoni[3], Yehuda Afek[3], Yaniv Harel[3], Jin Song Dong[2]

Shanghai Jiao Tong University[1], National University of Singapore[2], Tel Aviv University[3]

A Artifact Appendix

A.1 Abstract

Visual CAPTCHAs present users with interactive puzzles that must be solved to access protected online content. Their effectiveness relies on the assumption that these challenges are difficult for bots but easy for humans. However, the rise of general-purpose AI models (e.g., ChatGPT) calls this assumption into question, potentially undermining the reliability of current CAPTCHA systems.

To evaluate the security of visual CAPTCHAs in this new landscape, we implement our main artifacts: (1) Halligan, a generalized visual language model (VLM)-based CAPTCHA solver, and (2) an interactive offline benchmark containing 26 diverse types of visual CAPTCHAs. Halligan functions as a VLM agent equipped with tools to abstract the CAPTCHA layout, explore possible options, and reason about or compare individual choices. It automatically formulates each visual CAPTCHA as a search optimization problem and generates an executable Python script to solve it. This script is reusable across all instances of the same CAPTCHA type. For functionality, we present Halligan's script generation workflow through execution traces and conduct attacks using the generated scripts on selected samples from the benchmark.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Halligan could be misused to attack CAPTCHAs on real-world websites. Malicious actors might exploit it for purposes such as promotion spam, registration abuse, data scraping, or bot-driven raiding. Such misuse can undermine trust in online communities, degrade user experiences, and disrupt the normal operations of websites. However, in this AE, we restricted all attacks to our offline benchmark. The benchmark is self-contained and does not interact with any live websites or CAPTCHA service providers. As such, AE reviewers do not have associated risks.

A.2.2 How to access

Both Halligan and the benchmark are available in our Zenodo repository (DOI: 10.5281/zenodo.15580922). Please use the latest version.

A.2.3 Hardware dependencies

We tested the functionality on a desktop computer with 16GB of GPU VRAM (Optional), 8GB of system RAM, and 16GB of available disk space.

A.2.4 Software dependencies

We recommend using Linux. Our setup was tested on Ubuntu 20.04.3 LTS with Pixi 0.47.0, CUDA 12.1 (Optional), Docker 24.0.7, and Docker Compose 2.21.0.

A.2.5 Benchmarks

The benchmark for AE is provided in the artifact.

A.3 Set-up

Before proceeding with the installation, please make sure that the following prerequisite software is installed.

1. Install Docker Desktop[1]
2. Install Pixi[2]

A.3.1 Installation

Setup Benchmark

1. In a new terminal, download and unzip `benchmark.zip` from Zenodo.
2. Change into the `benchmark` directory:

[1] https://docs.docker.com/compose/install/
[2] https://pixi.sh/dev/installation/

3. Create and start containers.

   ```
   docker compose up
   ```

   This takes a few minutes. It begins by building the `benchmark` Docker image using the provided Dockerfile, which should have a hash ID `1a27e353ef13`. Once the build is complete, it will launch two containers: the benchmark server and a browser. These containers expose ports 3000 and 5000 on the host, respectively.

Setup Halligan

1. In a new terminal, download and unzip `halligan.zip` from Zenodo.
2. Change into the `halligan` directory.
3. Setup Pixi environment.

   ```
   pixi install
   ```

   This may take a few minutes for Pixi to resolve all installation dependencies, update the lockfile and install the environment.

4. Make a copy of `.env.example` with the filename `.env`.

   ```
   cp .env.example .env
   ```

5. In `.env`, replace the value of `OPENAI_API_KEY` with our provided key (no line breaks):

   ```
   sed -i "s/^OPENAI_API_KEY=.*/OPENAI_API_KEY=
   sk-proj-0G5YTPRtbreaTJSxjzbDtvvvgIPcZm4Ag3-
   WwuVusHBp1Wp4Z-Ws7mSSP44hYT7fTLdKY07M3DT3Bl
   bkFJZEnH8-BeqmYNOtxP47to2IMuhfxwagw3ZkUpy4fLF1
   pttROQiYfKCJ0ry0L4QRGaFClblcK74A/" .env
   ```

A.3.2 Basic Test

1. In a new terminal, navigate to `/halligan`
2. Run the script `basic_test.py`, which checks that the benchmark, browser, and the key components of Halligan are functioning correctly.

   ```
   pixi run pytest basic_test.py --verbose
   ```

   Once completed, verify that the terminal displays 29 lines of `PASSED` test messages:

   ```
   basic_test.py::test_browser PASSED
   basic_test.py::test_benchmark PASSED
   basic_test.py::test_captchas[...] PASSED
   ...
   basic_test.py::test_halligan PASSED
   ===== 29 passed in 31.45s =====
   ```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Halligan automatically generates an executable Python script for solving challenges of a given visual CAPTCHA type. This capability is demonstrated by Experiment (E1), whose three-stage process is detailed in Section 3 and illustrated in Figure 2.*

(C2): *Using the generated scripts, Halligan successfully solves at least one challenge from each of the 26 visual CAPTCHA types included in the benchmark. This is proven by Experiment (E2), with full details provided in Table 4.*

A.4.2 Experiments

(E1): *[10 human-minutes + 20 compute-minutes]: The first experiment demonstrates Halligan's script generation process.*

Preparation: It is assumed that you have followed the set-up instructions (§A.3.1)

Execution: Run the experiment as follows:

1. Open a new terminal session and navigate to `/halligan`.
2. Run the generation script:

   ```
   pixi run python generate.py
   ```

Results: Verify that a log file matching the pattern agent-*.log is created. In the log file, confirm that Halligan has generated scripts for 26 out of 26 CAPTCHAs, each with a unique name, and that each entry is followed by `[Stage 1] ... [Stage 3]`, resulting in a total of 104 lines. A sample output is shown below for reference:

```
Generating script (1 out of 26): lemin
    [Stage 1] Objective Identification
    [Stage 2] Structure Abstraction
    [Stage 3] Solution Composition
Generating script (2 out of 26): geetest/slide
    ...
Generating script (26 out of 26): amazon
    [Stage 1] Objective Identification
    [Stage 2] Structure Abstraction
    [Stage 3] Solution Composition
```

Next, verify that a new directory `/results/generation`, relative to the generation script, is created. This directory should contain 26 `.ipynb` generation trace files, each with a unique name. These files show how Halligan generates the solution script for each CAPTCHA.

```
/lemin.ipynb
/geetest_slide.ipynb
...
/amazon.ipynb
```

Examine one of the `.ipynb` trace files, verify that there is a `PROMPT =` and `RESPONSE =` code block under the headings "Objective Identification", "Structure Abstraction", and "Solution Composition". For example:

```
PROMPT = ...
```

```
RESPONSE = ...
TIME = 7.902754155918956
FINGERPRINT = fp_5bf33c1ec4
TOTAL_TOKENS = 866
PROMPT_TOKENS = 707
COMPLETION_TOKENS = 159
```

(E2): *[5 human-minutes + 20 compute-minutes]: The second experiment uses scripts generated by Halligan to solve 26 different types of CAPTCHAs in the benchmark.*
Preparation: It is assumed that you have followed the set-up instructions (§A.3.1)
Execution: Run the experiment as follows:

1. Open a new terminal session and navigate to `/halligan`.
2. Run the execution script:
   `pixi run python execute.py`

Results: Verify that a log file matching the pattern `agent-*.log` is created. In the log file, confirm that Halligan has tested 26 out of 26 CAPTCHAs, each with a unique name, and that each entry is followed by `Solved: True`, resulting in a total of 52 lines. A sample output is shown below for reference:

```
Testing CAPTCHA (1 out of 26): lemin
Solved: True
Testing CAPTCHA (2 out of 26): geetest/slide
Solved: True
...
Testing CAPTCHA (26 out of 26): amazon
Solved: True
```

Next, verify that a new directory `/results/execution`, relative to the execution script, is created. This directory should contain 26 `.ipynb` execution trace files, each with a unique name. These files demonstrate how Halligan's script derives the solution for each CAPTCHA.

```
/lemin.ipynb
/geetest_slide.ipynb
...
/amazon.ipynb
```

A.5 Notes on Reusability

Each CAPTCHA type in the benchmark is implemented as a Flask Blueprint and can be accessed or extended via routes following the pattern `.../{CAPTCHA_NAME}/{CHALLENGE_ID}`. For more information on Flask Blueprints, see this tutorial.

Halligan can be applied to other visual CAPTCHAs outside of the benchmark, without requiring additional configuration. For more details, see Section 4.4. Halligan also supports two example practical use cases: (1) it can aid in anti-phishing and security crawling by revealing CAPTCHA-cloaked phishing websites, and (2) it can assist webmasters and security practitioners in designing more robust anti-bot mechanisms by testing the effectiveness of new CAPTCHA challenges.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: <Topic-FlipRAG: Topic-Orientated Adversarial Opinion Manipulation Attacks to Retrieval-Augmented Generation Models>

Yuyang Gong[1]*, Zhuo Chen[1], Jiawei Liu[1]†, Miaokun Chen[1],
Fengchang Yu[1], Wei Lu[1], XiaoFeng Wang[2], Xiaozhong Liu[3]
[1]Wuhan University, [2]Nanyang Technological University, [3]Worcester Polytechnic Institute

A Artifact Appendix

A.1 Abstract

*This artifact contains the complete implementation of the **Topic-FlipRAG** attack, along with all components necessary to reproduce the results presented in our USENIX Security paper. The repository includes: (1) the full attack codebase for Topic-FlipRAG and baseline methods, (2) evaluation scripts for computing opinion shift and retrieval consistency metrics, (3) ethically filtered synthetic datasets generated by GPT-4o, (4) comprehensive documentation for setup and usage, and (5) pre-generated poisoned document examples for quick evaluation. These resources enable reproducibility.*

A.2 Description & Requirements

The artifact is fully executable via Google Colab and does not require local installation. All experimental code is configured with default parameters that exactly match those used in the main experiments of our USENIX Security paper. No changes are needed to reproduce the core results.

The repository includes pre-generated adversarial documents to facilitate rapid evaluation. Reviewers may directly run the evaluation notebooks to verify the effects of Topic-FlipRAG without executing the full document poisoning pipeline, thereby reducing time and API usage.

A.2.1 Security, privacy, and ethical concerns

Describe any risk for evaluators while executing your artifact to their machines security, data privacy or others ethical concerns. This is particularly important if destructive steps are taken or security mechanisms are disabled during the execution.

A.2.2 How to access

All artifact code, data, and documentation are publicly available at the following locations:

- **GitHub repository (for AEC evaluation and iterative updates):** `https://github.com/LauJames/Topic-FlipRAG`
- **Zenodo archive (stable version for the camera-ready submission):** `https://doi.org/10.5281/zenodo.15523434`

The GitHub repository is actively maintained during the Artifact Evaluation period and may receive minor updates in response to reviewer feedback. The Zenodo snapshot will serve as the final archived version for the camera-ready artifact.

A.2.3 Hardware dependencies

The artifact runs entirely on Google Colab without the need for any specialized local hardware or remote infrastructure.

- **GPU:** NVIDIA T4 (for generating poisoned documents via Topic-FlipRAG), A100 (recommended for evaluating RAG system response)

All components execute within Colab with GPU acceleration enabled; no SSH access or external hardware setup is required.

A.2.4 Software dependencies

The artifact runs in the default Google Colab environment using Python 3.9+. All required dependencies are open-source and automatically installed via `pip` at the beginning of each notebook.

*Email: `2498002636gyy@gmail.com`
†Corresponding author. Email: `laujames2017@whu.edu.cn`

Benchmarks

The artifact includes the dataset used for the opinion manipulation experiments. These consist of the PROCON dataset and GPT-4o–generated queries designed to simulate realistic manipulation tasks. All queries have been ethically filtered to ensure responsible usage and safe evaluation.

In addition, we provide pre-generated poisoned document examples used in our Topic-FlipRAG attack pipeline. These are included to help AEC reviewers quickly evaluate manipulation outcomes without running the full document poisoning process, thereby reducing computational cost and API usage.

A.3 Set-up

A.3.1 Installation

No manual installation is required. All dependencies are installed automatically when executing the code blocks in the provided Colab notebooks.

A.3.2 Basic Test

To perform a quick test, please follow the instructions in the GitHub repository's `README.md`. Replace the relevant path variables with the provided example data files. Default parameters have been pre-configured in the Colab notebooks, so reviewers can directly run the cells without any modification.

A.4 Evaluation workflow

The evaluation workflow is provided in `RAG_pipeline.ipynb`, which includes code for assessing both ranking manipulation and opinion manipulation. Reviewers can directly execute this notebook in Google Colab to reproduce the main experimental results described in the paper.

A.4.1 Major Claims

(C1): *Topic-FlipRAG, under black-box conditions, effectively boosts the rankings of target documents across queries within the retriever module of RAG. This is demonstrated by experiment (E1,E2,E3) described in Section 6.1, with results shown in Table 2.*

(C2): *Topic-FlipRAG significantly affects the answers generated by the target RAG systems. This is supported by experiment (E1,E2,E3) described in Section 6.2, with results shown in Table 3 and Table 11 (Appendix).*

A.4.2 Experiments

(E1): *[Knowledge-Guided attack] [5 human-minutes + 45 compute-minutes per topic + OpenAI API + NVIDIA T4]: This stage corresponds to generating poisoned documents (doc_know) aligned with the target opinions, as described in Section 4.2. These are used in subsequent trigger optimization and evaluation steps.*
How to: *We recommend testing on a single topic to reduce API costs and runtime. The generated doc_know will serve as input to Stage 2. Even if this step is skipped, subsequent evaluation can still be performed using our pre-generated example data.*
Preparation: *Open Stage 1 Notebook in Google Colab. Ensure GPU (T4) is enabled. Install dependencies via the first code cell. Place `PROCON_data.json` from the GitHub repo in the required data path. An OpenAI API key is also needed.*
Execution: *Run the notebook as instructed. Select a single topic and execute the document generation blocks.*
Results: *The notebook will output a doc_know JSON file for the selected topic, which can be saved to Colab or Google Drive. This file is used as input to Stage 2 (E2).*

(E2): *[Adversarial Trigger Generation] [5 human-minutes + 75 compute-minutes per topic (5 passages per topic, 42 topics total) + NVIDIA T4]: This stage corresponds to generating adversarial triggers based on the doc_know files produced in Stage 1 (E1), as described in Section 4.3 of the paper. The output is a poisoned document file (doc_adv) used in final RAG evaluation.*
How to: *Follow the GitHub `README.md` to run the corresponding Colab notebook for Stage 2. Even if E1 is not executed, we provide example doc_know files that can be used to run E2 directly.*
Preparation: *Open Stage 2 Notebook in Colab. Enable GPU (T4), install dependencies, and ensure `PROCON_data.json` is placed in the correct path. For quick testing, update the input path to use the provided example doc_know file, as explained in the README.*
Execution: *Run all cells in the notebook. The script will optimize adversarial triggers for five passages per topic based on the input doc_know.*
Results: *The output is a doc_adv file containing adversarially modified documents, stored locally or in Drive. This file will be used as input in Stage 3 (E3) to evaluate attack effectiveness.*

(E3): *[RAG Pipeline and Evaluation] [5 human-minutes + 3 compute-minutes per topic + A100 GPU + OpenAI API]: This stage involves injecting adversarial documents into the RAG system and evaluating downstream impacts, as described in Section 6. This step validates Claims C1 and C2 through end-to-end experiments.*
How to: *Follow the instructions in the GitHub `README.md` to run the `RAG_pipeline.ipynb` notebook.*
Preparation: *Launch the notebook in Google Colab and ensure GPU (A100) is enabled. Install dependencies by running the setup cells. Place*

`PROCON_data.json` in the correct path. To simplify evaluation, we provide pre-generated poisoned passages (doc_adv) for 9 topics in the "Society" domain with target stance "CON." Update the input path as described in the README.

Execution: *Run all cells in the notebook. This step loads the poisoned documents, runs the retrieval-augmented generation process, and computes evaluation metrics.*

Results: *The expected metrics, based on our paper results using the provided poisoned examples (Society domain, target stance: CON), are: RASR = 51.9, Top3-v = 25.57, and ASV = 0.55 (ranging approximately from 0.5 to 0.6). These results were obtained using the **Qwen2.5** LLM with a **Contriever** retriever setup, and demonstrate the manipulation effectiveness of Topic-FlipRAG in a full RAG pipeline.*

If you wish to evaluate other combinations of LLMs and retrievers, please modify the corresponding configuration parameters in the notebook. Refer to the experimental comparisons reported in the main paper for baseline expectations across different model setups.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Sound of Interference: Electromagnetic Eavesdropping Attack on Digital Microphones Using Pulse Density Modulation

Arifu Onishi*
The University of Electro-Communications

S. Hrushikesh Bhupathiraju*
University of Florida

Rishikesh Bhatt
University of Florida

Sara Rampazzi
University of Florida

Takeshi Sugawara
The University of Electro-Communications

A Artifact Appendix

A.1 Abstract

As described in the paper, we discovered a new side-channel attack on pulse density modulation (PDM) digital MEMS microphones which are used in common laptops and IoT devices. In the paper, discover that each harmonic of these digital pulses retains acoustic information, allowing the original audio to be retrieved through simple FM demodulation using standard radio receivers, even in behind-the-wall scenarios. In this artifact appendix, we provide spoken digits and spoken sentence audio data reconstructed by the attack across various devices using a 14-element Yagi antenna, a magnetic loop antenna, and a cheap antenna made from copper foil. The artifacts also include the scripts to fine-tune and evaluate the HuBERT transformer model for digit and speaker recognition and evaluate the data with HuBERT, OpenAI, and Microsoft speech transcription models required to reproduce the results described in the paper (more specifically, the outcome of the analysis depicted in Tables 4, 5, 6, 7, and 8 of the paper).

A.2 Description & Requirements

The artifact contains scripts (and models) to fine-tune Transformer models for speech recognition, API scripts for inferring and evaluating speech transcription.

A.2.1 Security, privacy, and ethical concerns

Our artifact is non-destructive, and there is no security or privacy risk to the evaluators when the training, fine-tuning, and inference models.

A.2.2 How to access

Our research artifacts, including the speech transcription results and training log files for digit and speaker classification models, along with the corresponding recorded audio files for the tested devices and the scripts for training and fine-tuning the pipeline we used for our speech recognition and transcription models are available at: https://zenodo.org/records/14736347.

We include a README file in the repository to guide the reviewers through the setup of the environment, and the information regarding the provided data corresponding to the experiment in the paper.

A.2.3 Hardware dependencies

We build our recognition models by fine-tuning HuBERT transformer models on our dataset. To effectively fine-tune HuBERT models, you typically need a GPU with at least 16GB of VRAM. Our model was fine-tuned on a single Nvidia A100 GPU (40 GB), with a fine-tuning duration of approximately 45 minutes.

A.2.4 Software dependencies

There is no dependency on the Operating System used by the user. To run the inference codes, the user needs access to Python 3 (above 3.8), conda environment with the following Python packages: pytorch==2.4.0, torchvision==0.19.0, torchaudio==2.4.0, pytorch-cuda=11.8, pandas, scipy, tqdm, matplotlib, seaborn, scikit-learn, transformers, openpyxl, PyWavelets, librosa, datasets, openai, and azure-cognitiveservices-speech.

Detailed instructions to setup the conda environment and the python libraries are included in the README file available at: https://zenodo.org/records/14736347. Note that running the OpenAI and Microsoft APIs for speech transcription models requires paid subscriptions and the API keys. The HuBERT transcription model on the other hand can be implemented and evaluated using the provided scripts.

A.2.5 Benchmarks

All the data required for the artifacts are available at the provided link.

A.3 Set-up

A.3.1 Installation

Run the following commands to configure the conda environment.

- conda create -y –name SoI python==3.8.19
- conda activate SoI
- conda install pytorch==2.4.0 torchvision==0.19.0 torchaudio==2.4.0 pytorch-cuda=11.8 -c pytorch -c nvidia
- pip install pandas scipy tqdm matplotlib seaborn scikit-learn transformers openpyxl PyWavelets librosa
- pip install datasets
- pip install openai
- pip install azure-cognitiveservices-speech
- pip install jiwer

The test the performance of the attack, download the reconstructed audio from https://zenodo.org/records/14736347 and run the python scripts included in the built conda environment.

A.3.2 Basic Test

To evaluate the Word Error Rates (WER) transcription results from the CSV files provided, run *Evaluation_Results/Transcription/evaluation_wer.py* by changing the CSV file path to the evaluation experiment. To test the fine-tuning on digit recognition by the running the following command: *python3 HuBERT_emispeech_pre_trained_digit_Classifier.py –json_file sample.json* by changing the path in sample.json file to the following:
'/Feasibility/FSDD'.

To run the speaker recognition, replace 'digit' in the script name with 'speaker' and run the following command: *python3 HuBERT_emispeech_pre_trained_speaker_Classifier.py –json_file sample.json*

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The results shown in Tables 4, 5, 6, 7, and 8 for digit and speaker recognition can be achieved by fine-tuning the HuBERT model on the FSDD dataset reconstructed from the attack (provided in this artifact).

(C2): The speech transcription results shown in Table 4, 5, and 8 can be reproduced by running the transcription_hubert.py, transcription_OpenAI.py, and transcription_mirosoft.py scripts on the corresponding data as explained in A.4.2.

A.4.2 Experiments

(E1): *[Generality across off-the-shelf PDM Microhpones] [10 human-minutes + 5 compute-hour]: This experiment reproduces the results shown in Table 4*
How to: In this experiment we run the FSDD and Harvard Sentences data collected from different PDM microphones to achieve speech recognition and transcription respectively.
Preparation: To get the fine-tune digit and speaker recognition results, edit the path for the dataset in the sample.json file with the following directories

- '/PDMs/Knowles/FSDD'
- '/PDMs/STM/FSDD'
- '/PDMs/TDK-41350/FSDD'
- '/PDMs/TDK-T3902/FSDD'
- '/PDMs/VM3000/FSDD'

To run speech transcription using HuBERT, OpenAI, and Microsoft Speech Studio, change the directory path in *transcription_hubert.py, transcription_OpenAI.py, and transcription_mirosoft.py* scripts with the below paths iteratively.

- '/PDMs/Knowles/Harvard_Sentences'
- '/PDMs/STM/Harvard_Sentences'
- '/PDMs/TDK-41350/Harvard_Sentences'
- '/PDMs/TDK-T3902/Harvard_Sentences'
- '/PDMs/VM3000/Harvard_Sentences'

Execution: Run the following script to fine-tune the digit and speaker recognition models: *python3 HuBERT_emispeech_pre_trained_digit_Classifier.py –json_file sample.json*. (Replace 'digit' in the file name with 'speaker' to run fine-tuning on speaker recognition) Run the following commands to run the speech transcription models

- *python transcription_hubert.py*
- *python transcription_OpenAI.py*
- *python transcription_microsoft.py*

Results: The results for the fine-tuning models will be printed on the console. The results can also be saved into a log file to check the final model performance. Note that the classification accuracy can vary slightly due to randomized train-val split. The results from the transcription scripts will be saved to csv files with the audio file name and the corresponding transcription results. *Evaluation_Results/Transcription/evaluation_wer.py* can be used to extract the Word Error Rate (WER) from transcription results in CSV files.

(E2): *[Behind the Wall Scenarios] [10 human-minutes + 5 compute-hour]: This experiment reproduces the results shown in Table 5*

How to: In this experiment we run the FSDD and Harvard Sentences data collected from behind-the-wall scenarios.

Preparation: To get the fine-tune digit and speaker recognition results, edit the path for the dataset in the sample.json file with the following directories

- '/Behind_the_wall/Audio_Microphone/FSDD'
- '/Behind_the_wall/Loop/15cm/FSDD'
- '/Behind_the_wall/Loop/20cm/FSDD'
- '/Behind_the_wall/Loop/25cm/FSDD'
- '/Behind_the_wall/Copper/15cm/FSDD'
- '/Behind_the_wall/Copper/20cm/FSDD'
- '/Behind_the_wall/Copper/25cm/FSDD'

To run speech transcription using HuBERT, OpenAI, and Microsoft Speech Studio, change the directory path in *transcription_hubert.py, transcription_OpenAI.py, and transcription_mirosoft.py* scripts with the below paths iteratively.

- '/Behind_the_wall/Audio_Microphone/Harvard_Sentences'
- '/Behind_the_wall/Loop/15cm/Harvard_Sentences'
- '/Behind_the_wall/Loop/20cm/Harvard_Sentences'
- '/Behind_the_wall/Loop/25cm/Harvard_Sentences'
- '/Behind_the_wall/Copper/15cm/Harvard_Sentences'
- '/Behind_the_wall/Copper/20cm/Harvard_Sentences'
- '/Behind_the_wall/Copper/25cm/Harvard_Sentences'

Execution: Run the following script to fine-tune the digit and speaker recognition models: *python3 HuBERT_emispeech_pre_trained_digit_Classifier.py –json_file sample.json*. (Replace 'digit' in the file name with 'speaker' to run fine-tuning on speaker recognition) Run the following commands to run the speech transcription models

- *python transcription_hubert.py*
- *python transcription_OpenAI.py*
- *python transcription_microsoft.py*

Results: The results for the fine-tuning models will be printed on the console. The results can also be saved into a log file to check the final model performance. Note that the classification accuracy can vary slightly due to randomized train-val split. The results from the transcription scripts will be saved to csv files with the audio file name and the corresponding transcription results. *Evaluation_Results/Transcription/evaluation_wer.py* can be used to extract the Word Error Rate (WER) from transcription results in CSV files.

(E3): *[Long Distance Experiments] [10 human-minutes + 5 compute-hour]: This experiment reproduces the results shown in Table 6*

How to: In this experiment we run the FSDD data collected from long distances in behind-the-wall scenarios.

Preparation: To get the fine-tune digit and speaker recognition results, edit the path for the dataset in the sample.json file with the following directories

- '/Long_Distance/1m'
- '/Long_Distance/2m'
- '/Long_Distance/3m'
- '/Long_Distance/4m'

Execution: Run the following script to fine-tune the digit and speaker recognition models: *python3 HuBERT_emispeech_pre_trained_digit_Classifier.py –json_file sample.json*. (Replace 'digit' in the file name with 'speaker' to run fine-tuning on speaker recognition)

Results: The results for the fine-tuning models will be printed on the console. The results can also be saved into a log file to check the final model performance. Note that the classification accuracy can vary slightly due to randomized train-val split.

(E4): *[Room Scenario Evaluation] [10 human-minutes + 5 compute-hour]: This experiment reproduces the results shown in Table 7*

How to: In this experiment we run the FSDD data collected from long distances in behind-the-wall scenarios.

Preparation: To get the fine-tune digit and speaker recognition results, edit the path for the dataset in the sample.json file with the following directories

- '/Long_Distance/1m'
- '/Room_Scenarios/Concrete_wall'
- '/Room_Scenarios/Oriented'
- '/Room_Scenarios/Through_Monitor'

Execution: Run the following script to fine-tune the digit and speaker recognition models: *python3 HuBERT_emispeech_pre_trained_digit_Classifier.py –json_file sample.json*. (Replace 'digit' in the file name with 'speaker' to run fine-tuning on speaker recognition)

Results: The results for the fine-tuning models will be printed on the console. The results can also be saved into a log file to check the final model performance. Note that the classification accuracy can vary slightly due to randomized train-val split.

(E5): *[Automated Fine-tuning Evaluation] [10 human-minutes + 20 compute-hour]: This experiment will automatically reproduce all the fine-tuning results discussed until now.*

How to: In this experiment we automatically run the FSDD fine-tuning on all the data.

Preparation: To get the fine-tuned digit and speaker recognition results, prepare the data as follows:

- 'unzip Evaluation_Scripts.zip'
- 'cd Evaluation_Scripts'
- 'chmod +x evaluate.sh'

Execution: Run the following command to fine-tune and test the digit and speaker recognition models for all the data: *'./evaluate.sh'*.

Results: Upon running evaluate.sh, all compressed FSDD data directories are automatically extracted into the same directory as the script. Following extraction, the digit and speaker recognition pipelines are executed sequentially. Each pipeline generates a corresponding results directory—suffixed with either _digit or _speaker—within the same path. These result folders contain all relevant artifacts, including fine-tuning logs, training and testing metrics, confusion matrices, and model checkpoints.

(E6): *[Automated Transcription Evaluation] [10 human-minutes + 2 compute-hour]: This experiment will automatically reproduce all the transcription results discussed until now.*

How to: In this experiment we automatically run the speech transcription on all the data.

Preparation: To get the speech transcription, prepare the data as follows:

- 'cd Evaluation_Scripts'
- 'chmod +x evaluate_transcription.sh'

Execution: Run the following command to get transcription for all the Harvard_Sentences data: *'./evaluate_transcription.sh'*.

Results: Upon running evaluate_transcription.sh, all relevant audio directories containing Harvard Sentences are processed from the same directory as the script. The transcription evaluation is performed sequentially using HuBERT (local), OpenAI Whisper API, and Microsoft Azure Speech API. Each backend generates a corresponding CSV output, prefixed with the backend name (e.g., hubert_, openai_, microsoft_) and suffixed with the name of the data directory. All transcription results are stored in a shared transcription_results folder alongside the script.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: COLLISIONREPAIR:First-Aid and Automated Patching for Storage Collision Vulnerabilities in Smart Contracts

Yu Pan[*†], Wanjing Han[*†], Yue Duan[‡], Mu Zhang[†]
[†]University of Utah, United States [‡]Singapore Management University, Singapore
[†]{yu.pan, wanjing.han, mu.zhang}@utah.edu [‡]yueduan@smu.edu.sg

A Artifact Appendix

A.1 Abstract

This artifact accompanies the paper "COLLISIONREPAIR: *First-Aid and Automated Patching for Storage Collision Vulnerabilities in Smart Contracts*". This repository contains scripts and results for evaluating smart contract patching. Detailed patching instructions are provided in `octopus/PATCHING.md`. The `README` file offers step-by-step guidance for running the evaluation.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

COLLISIONREPAIR is a system designed to patch upgradeable smart contract bytecode with storage collision vulnerabilities. All experiments in this study were conducted in controlled, sandboxed environments using Ganache with isolated snapshots of the Ethereum blockchain. As a result, traditional ethical concerns associated with cybersecurity research—such as experimentation on live systems, data privacy, or human subject involvement—were not applicable. The experiments posed no risk to real-world blockchain deployments or operations, even when executing vulnerable contracts or replaying attack transactions.

Importantly, COLLISIONREPAIR is a mitigation tool, not a detection mechanism—it reveals vulnerabilities only when an actual attack occurs. The vulnerable contracts evaluated in our study were sourced from the CRUSH dataset, where issues had already been disclosed and, in some cases, exploited. Our transaction replays confirm that COLLISIONREPAIR successfully blocks these previously observed attacks. No new vulnerabilities were discovered in the additional contracts we tested; should any new issues emerge, we will follow responsible disclosure practices, including notifying developers and CISA using publicly available contact information on Etherscan.

This research complies with all relevant legal and ethical standards. No additional regulatory approvals were required. Our work advances the field of blockchain and smart contract security without introducing new risks or ethical concerns.

A.2.2 How to access

https://doi.org/10.6084/m9.figshare.29150726

A.2.3 Hardware dependencies

- **Processor:** Any 64-bit processor, such as Intel Core Processor Series.
- **Memory:** 4 GB RAM or higher.
- **Storage:** 1 GB or more available disk space.

A.2.4 Software dependencies

Tool	Purpose
Node.js >= v16	For deploying contracts and running scripts
Python >= 3.10.16	For static verification and patching
Ganache	Local Ethereum testnet (must be running)
Octopus	Bytecode instrumentation engine

Table 1: Tool requirements and their purposes.

Software dependencies are listed in Table 1. For more details and relevant links, please refer to the README.

A.2.5 Benchmarks

Around 4,000 contracts were used to verify the correctness of patching and to evaluate transaction replay behavior after patching. Details of the project structure can be found in the `README` under the *Directory Structure* section.

Data The `contracts.txt` file lists all contract addresses to be processed. The `results/` directory stores all output data for each contract, including ABI, bytecode, transactions, and replay results.

[*]Yu Pan and Wanjing Han contributed equally to this work.

Scripts All main scripts are located in `evaluation/correctness/scripts/correctness/`. These include deployment, replay, patching, and result filtering scripts.

Core Patch Function The `patch.sh` script is provided to patch contracts in the correctness dataset. Run it from either the root directory or the `evaluation/correctness` directory to apply automated patching to all contracts listed in `contracts.txt`.

A.3 Set-up

A.3.1 Installation

Install Octopus Our patching tool is built on a modified version of `Octopus` (https://github.com/FuzzingLabs/octopus).

You can install it using either of the following methods:

```
cd octopus
sudo python3 setup.py install
cd -
```

or

```
cd octopus
pip install -r requirements.txt
cd -
```

Install Node.js Dependencies Navigate to the correctness evaluation directory and install the required Node.js packages:

```
cd evaluation/correctness
npm install
cd -
```

A.3.2 Basic Test

Verifying Patched Bytecode **Script:** `verify_patch.sh` `evaluation/scripts/verify_instrumentation.py`: This script takes a folder as input, automatically compares the original and patched bytecode, and displays the results.

```
chmod 777 verify_patch.sh
./verify_patch.sh
```

Expected outputs can be found in the `README` under the *Verifying Existing Results* section.

A.4 Evaluation workflow

A.4.1 Major Claims

COLLISIONREPAIR can successfully patch all contracts in the provided benchmark and can replay most real-world transactions both before and after patching in the Ganache environment.

A.4.2 Experiments

Applying the Patch Tool Before applying the patching tool, back up the existing `results/` directory to preserve previous outputs:

```
cp -r evaluation/correctness/results evaluation/
    correctness/results_backup
```

Deploy the StorageTracker Contract Ganache must be running at http://127.0.0.1:7545.
Deploy the monitoring contract and update the configuration:

```
node js_scripts/monitor/deploy_monitor.js
```

Run the Patching Script After deploying StorageTracker, apply the patch to all contracts in the dataset using the following commands:

```
chmod 777 patch.sh
./patch.sh
```

Example output:

```
INFO - [+] Runtime code detected
INFO - [+] Runtime code detected
WARNING - function signatures collision: ['symbol()',
     'link_classic_internal(uint64,int64)']
WARNING - function signatures collision: ['decimals
    ()', 'available_assert_time(uint16,uint64)']
...
```

Performance: Patching 4,000 contracts takes approximately 2 to 4 hours depending on hardware.

Regenerating transaction replay results is optional and detailed in the *Regenerating Dynamic Execution Results* section of the `README`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Does Finality Gadget Finalize Your Block? A Case Study of Binance Consensus

Rujia Li
Tsinghua University

Jingyuan Ding
Shandong University

Qin Wang
CSIRO Data61

Keting Jia
Tsinghua University

Haibin Zhang
Yangtze Delta Region Institute of Tsinghua University

Sisi Duan
Tsinghua University

A Artifact Appendix

A.1 Abstract

Fast Finality (FF) is a core mechanism of BNB Smart Chain (BSC). BSC claims that its FF mechanism can finalize blocks in $O(1)$ time, simultaneously reducing latency and improving stability. We present three attacks, showing BSC fails to finalize blocks in constant time and may even simply fail to achieve liveness. We implement attacks and evaluate the effect of the three attacks on a local-built testnet with 21 clients. The results match our theoretical analysis. This artifact aims to reproduce the results in Section 7 of our paper.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical concerns

Our experiments can be launched using a local testnet (using even *one* local machine). As reported in the paper (Section 1), we have done responsible disclosure to the Binance team. The team has confirmed and acknowledged all the attacks.

A.2.2 How To Access

Our artifact is openly available on GitHub at the fixed and stable commit https://github.com/tsinghua-cel/bsc-attack-experiment/tree/a40ca69ff22aa1b3a2ee61a109825c533044fb58. The repository contains all the scripts, container images, source codes, and sample output files. For long-term preservation, we have also archived an identical snapshot on Zenodo: https://doi.org/10.5281/zenodo.15852608.

A.2.3 Hardware Dependencies

The experiments do not require particular hardware. Our setup used a standard machine with an 8-core CPU, 32GB RAM, 500GB ROM storage, and a 100 Mbps internet connection.

Note: We recommend allocating at least **16GB** of memory to Docker to ensure the smooth execution of our experimental environment.

A.2.4 Software dependencies

We ran all experiments using Docker (version 27 or higher), which can be installed by following the official guide at https://docs.docker.com/engine/install/.

A.2.5 Benchmarks

Not applicable.

A.3 Set Up and Run Experiments

A.3.1 Installation

No additional setup is required. All dependencies and environment configurations are encapsulated in a Docker image to ensure consistent and reproducible results. For reviewers who are interested in customizing their local testing environment, detailed manual instructions along with raw test scripts and data are available at https://github.com/tsinghua-cel/bsc-attack-experiment/tree/a40ca69ff22aa1b3a2ee61a109825c533044fb58.

A.3.2 Pull Docker Images (One-time setup)

The following commands show how to pull the images for attack 1, attack 2, and attack 3 (involving two configurations).

```
docker pull erick785/bsc-attack-1:latest
docker pull erick785/bsc-attack-2:latest
docker pull
 ↪  erick785/bsc-attack-3-bootnode:latest
docker pull
 ↪  erick785/bsc-attack-3-staticnode:latest
```

A.3.3 Basic Test

Impact of the attacks:

- **attack 1**: Slowing down the finality of blocks.
- **attack 2**: No blocks can be finalized.
- **attack 3**: No blocks can be finalized for a period of time.

Table 1: Column meanings in datalog (results)

Column	Meanings	Description
1	The latest block height	The number of the latest block that has been generated by the current node (`LatestBlock`).
2	Finalized block height	Block number that has been finalized (`FinalizedBlock`).
3	Attestation of block header	`true` means that the block received a vote attestation, `false` means that it did not.

** All attacks start at block height 250 **

Reviewers can then run the artifact by executing:

For attack 1

- Run command:

```
touch 1.txt && docker run -it --rm -v
 ↪  ./1.txt:/app/query/21.txt
 ↪  erick785/bsc-attack-1:latest
```

- The following outputs are expected (cf. Table 1):

```
1,0,false
2,0,false
3,0,false
4,0,false   // votes not collected
...
210,0,false
211,0,false
212,0,true
213,211,true
214,212,true
...
```

The experiment for **attack 1** will be automatically terminated after 100 minutes of execution. The results generated by the Docker container will be saved to the file `1.txt`.

For attack 2

- Run command:

```
touch 2.txt && docker run -it --rm
 ↪  -v ./2.txt:/app/query/21.txt
 ↪  erick785/bsc-attack-2:latest
```

- The following outputs are expected:

```
1,0,false
2,0,false
3,0,false
4,0,false  // votes not collected
...
```

The experiment for **attack 2** will be automatically terminated after 100 minutes of execution. The results generated by the Docker container will be saved to the file `2.txt`.

For attack 3 (bootnode mode)

- Run command:

```
touch 3.1.txt && docker run -it --rm -v
 ↪  ./3.1.txt:/app/query/21.txt -e
 ↪  DELAY_INTERVAL_MS=25
 ↪  erick785/bsc-attack-3-bootnode:latest
```

- The following outputs are expected:

```
1,0,false
2,0,false
3,0,false
4,0,false  // votes not collected
...
```

The experiment for **attack 3 (bootnode mode)** will be automatically terminated after 100 minutes of execution. The results generated by the Docker container will be saved to the file `3.1.txt`. Notably, the `DELAY_INTERVAL_MS` field can be set to *25*, *50*, or *75*, indicating three separate experiments for the bootnode mode. We only present the case with 25-ms delay.

For attack 3 (full connection mode)

- Run command:

```
touch 3.2.txt && docker run -it --rm -v
 ↪  ./3.2.txt:/app/query/21.txt -e
 ↪  DELAY_INTERVAL_MS=25
 ↪  erick785/bsc-attack-3-staticnode:latest
```

- The following outputs are expected:

```
1,0,false
2,0,false
3,0,false
4,0,false  // votes not collected
...
```

The experiment for **attak 3 (full connection mode)** will be automatically terminated after 100 minutes of execution. The results generated by the Docker container will be saved to the file `3.2.txt`. Notably, the `DELAY_INTERVAL_MS` field can be set to *25*, *50*, or *75*, indicating three separate experiments for the full connection mode.

Note: Each execution launches a one-time Docker container. Once exited (e.g., via `Ctrl+C`), the container is removed and must be restarted from scratch. There is no need to delete the txt file (e.g., `1.txt`); re-running the command will overwrite the txt file with new results.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *In attack-I (attack 1), three blocks are finalized among 21 blocks, and the attack slows down the finality of blocks. This is proven by the experiment (E1), which reproduces the results described in Figure 8(a), Section 7.*

(C2): *In Attack-II (attack 2), the finalized block height remains unchanged, indicating that the system is unable to finalize new blocks and experiences a liveness failure. This is proven by the experiments (E2), which reproduce the results described in Figure 8(b), Section 7.*

(C3): *In attack-III (attack 3), we assess the **bootnode** mode and **full connection** mode of P2P. We set up the experiments so that Byzantine validators send their blocks 25 ms, 50 ms, and 75 ms ahead of time, and the results show that there exist some periods where no blocks can be finalized. In the extreme case, none of the 130 slots can be finalized (there is uncertainty in the outcome of the run). This is proven by the experiments (E3), which reproduce the results described in Figures 8(c)-8(f), Section 7.*

A.4.2 Experimental results

Overview: *In each experiment, we run more than 2,000 slots on our testnet and assess the fast finality and reward. Our attack is launched at the 250-th slot.*

(E1): For **attack 1**, the experiment is expected to finish in 100 minutes. However, for quicker testing, one may choose to run only 300 slots, which takes about 15 minutes.

Results: *During this experiment, the number of the latest block and finalized block is recorded in* `1.txt`. *An example of the txt file is as follows:*

```
1,0,false
2,0,false
3,0,false
4,0,false
...
210,0,false
211,0,false
212,0,true
213,211,true
214,212,true
...
247,245,true
248,246,true
249,247,true
250,248,true # Start attack.
251,248,false # Finalized block stops to increase
252,248,false
253,248,false
...
266,248,false
267,248,false
268,248,true
269,267,true
270,268,true # Finalized blocks catch up to 268
271,269,true # Repetitive pattern for below blocks
272,269,false
273,269,false
274,269,false
...
289,270,false
290,288,true
291,289,true
292,290,true
293,290,false
...
```

The three data fields are: The latest block height, finalized block height, and presence of attestation.

Note: This experiment is probabilistic. The exact data may vary across runs. However, the observed trend remains consistent: before the attack is launched, the difference between the latest block height and the finalized block height consistently stays at 2(e.g., [247,245], [248, 246], [249, 247]..). After initiating the attack at slot 250, this gap begins to widen. The finality of blocks no longer progresses in step with the increasing latest block height. This observation aligns with our claims in Table 1 and Figure 8(a): Attack 1 slows down the finality of blocks.

(E2): For **attack 2**, the experiment is expected to finish in 100 minutes. For quicker testing, one may choose to run only 300 slots, which takes about 15 minutes.

Results: *During this experiment, the number of the latest block and finalized block is recorded in* `2.txt`. *The fields in the txt file are the same as in **E1**. The output of the txt file is as follows:*

```
...
210,0,false
211,0,false
212,0,true
213,211,true
214,212,true
...
248,246,true
249,247,true
250,248,true # Start attack
251,248,false # Finalized blocks stall
252,248,false
253,248,false
254,248,false
255,248,false
256,248,false
...
```

Before launching the attack, the difference between the latest block height and the finalized block height consistently remained at 2. After we launch the attack at slot 250, we can observe that the last finalized block height remains fixed at 248, indicating that none of the subsequent blocks can receive more than $N - f$ matching attestations. As a result, no new block can be finalized. This observation supports our claims in Table 1 and Figure 8(b): Attack 2 causes a liveness failure.

(E3): For **attack 3**, we consider both bootnode mode and full connection mode, with each mode supporting configurable delay durations. For quicker testing, one may run only 300 slots, which takes around 15 minutes.

——- E3.1 bootnode mode ———-

Results: *During this experiment, the number of the latest block and finalized block is recorded in* `3.1.txt`. *The fields in the txt file are the same as in **E1**. The output of the txt file is as follows:*

```
...
432,429,false
433,429,true
434,429,false
435,429,false
...
445,429,false
446,429,false
447,429,false
448,429,false
449,429,false
450,429,false # At block 450, the finalized
block height remains 429 (21 blocks delayed)
451,429,false
452,429,false
...
470,429,false
471,429,false
472,429,false
473,429,true
474,429,false
475,429,true
476,474,true # After a delay, the finalized
block starts catching up
477,474,false
478,474,true
479,477,true
...
```

Note: This experiment is probabilistic. The exact data may vary from run to run. However, the observed trend remains consistent: After launching attack-III, the finalized blocks no longer increase in synchronization with the latest blocks, but rather increase in a delayed manner. This observation supports our claims in Table 3 and Figure 8(c)-8(e), Section 7 that attack-III prevents blocks from being finalized for a period of time.

——- E3.2 full node mode ———-

Results: *During this experiment, the number of the latest block and finalized block is recorded in* `3.2.txt`. *The fields in the txt file are the same as in **E1**. The output of the txt file is as follows:*

```
...
210,0,false
211,0,false
212,0,true
213,211,true
214,212,true
...
651,632,false
652,632,false
653,632,false
654,632,false
655,632,false
656,632,true
657,632,false
658,632,true
659,657,true
660,657,false
661,657,false
662,657,true
663,657,false
664,657,true
665,663,true
666,663,false
667,663,true
668,666,true
669,667,true
670,667,false
671,667,true
...
```

Note: This experiment is probabilistic. The exact data varies from run to run. However, the observed trend remains consistent: after launching attack 3, the finalized blocks no longer increase in synchronization with the latest blocks, but rather increase in a delayed manner. This observation supports our claims in Figure 8(d) and 8(f) that attack-III prevents blocks from being finalized for a period of time.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Recover from Excessive Faults in Partially-Synchronous BFT SMR

Tiantian Gong
Purdue University

Gustavo Franco Camilo
Purdue University

Kartik Nayak
Duke University

Andrew Lewis-Pye
LSE

Aniket Kate
Purdue University / Supra Research

A Artifact Appendix

A.1 Abstract

Byzantine fault-tolerant (BFT) state machine replication (SMR) protocols form the basis of modern blockchains as they maintain a consistent state across all blockchain nodes while tolerating a *bounded number* of Byzantine faults. We analyze BFT SMR in the **excessive fault** setting where the actual number of Byzantine faults surpasses a protocol's tolerance and in partial synchrony where the network experiences a known bounded delay after an unknown global stabilization time (GST). We devise the very first **recovery algorithm** for linearly chained and quorum-based partially synchronous SMR to recover from faulty states (i.e., correct replicas having equivocating states) caused by excessive faults.

We implement and evaluate the recovery procedure for HotStuff, addressed as "recover-HotStuff", in Rust. During implementation, we first test the performance of recover-HotStuff against vanilla HotStuff in the fault-free setting. The throughput resumes to the normal level (without excessive faults) after recovery routines terminate for 7 replicas and is slightly reduced by $\leq 4.3\%$ for 30 replicas. The latency is increased by 12.87% for 7 replicas and 8.85% for 30 replicas on average.

We then evaluate the performance of recover-HotStuff in the excessive fault setting. Each Byzantine replica is simulated with multiple consensus instances where it sends contradicting messages to honest replicas via different instances. We demonstrate that the end-to-end latency and throughput resume to normal level after recovery terminates. We then assess how the average recovery time changes according to different GST. The results indicate that the recovery time is roughly logarithmic in the length of contradicting chains (which is determined by GST).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The execution of the artifact poses no risks to the evaluators' machines, data privacy, or security. The evaluation is conducted within a controlled environment, specifically our own Byzantine Fault-Tolerant (BFT) system. The equivocation is simulated and executed solely on this system, which is isolated from external networks.

A.2.2 How to access

A stable version of the artifact is available at https://github.com/gFrancoCamilo/fault-free-recovery/tree/c9326eb07d347702b898e4c5910a44b3a86986a3 for the fault free setting and at https://github.com/gFrancoCamilo/one-shadow-recovery/tree/781e8b06145bf13c3ef53d54b325dca6c06fcc5b for the one shadow setting. A zip file containing both settings can also be found at https://zenodo.org/records/15133737.

A.2.3 Hardware dependencies

For optimal performance, we estimate that each client or node require at least one core. As a result, a machine with a minimum of 64 cores is recommended to reproduce the results. We also recommend at least 16 GB of RAM, an NVMe SSD, and a minimum of 30 GB to handle the code and the result files.

A.2.4 Software dependencies

Our code requires a Linux-based OS with bash. No specific distribution or version is required. Our experiments were run on Ubuntu 22.04 LTS. To compile and run our artifact on a local machine, the user must install the following softwares:

- Rust and Cargo: https://doc.rust-lang.org/cargo/getting-started/installation.html
- Python 3.6: https://www.python.org/downloads/
- Tmux: https://github.com/tmux/tmux/wiki/Installing
- Clang: https://clang.llvm.org/

In Ubuntu 22.04, the user can install the requirements by running:

```
$ sudo apt update
$ sudo apt-get install -y python3 tmux
clang python-is-python3 curl python3-pip git
$ curl https://sh.rustup.rs -sSf | sh
```

Make sure that `cargo` is in your `$PATH` after installation:

```
$ source $HOME/.cargo/env
```

In case Ubuntu 22.04 is not available, we recommend using Podman or Docker to set up an Ubuntu 22.04 container:

- Podman: https://podman.io/

Once podman is installed, the user should run:

```
$ podman pull docker://ubuntu:22.04
$ podman run -it ubuntu:22.04 /bin/bash
$ apt update
$ apt-get install -y python3 tmux clang
python-is-python3 curl python3-pip git
$ curl https://sh.rustup.rs -sSf | sh
```

Finally, make sure that `cargo` is in your `$PATH` after installation:

```
$ source $HOME/.cargo/env
```

A.2.5 Benchmarks

Our artifact uses require a synthetic workload of transactions to evaluate the performance of the system. Each transaction is a fixed-size byte sequence, with a size specified by the user. Therefore, no external benchmarks or datasets are required for the experiments.

A.3 Set-up

A.3.1 Installation

You can get the code by cloning our git repositories:

```
$ git clone https://github.com/gFrancoCamilo/
one-shadow-recovery.git
$ git clone https://github.com/gFrancoCamilo/
fault-free-recovery.git
```

We also require a few additional Python libraries that assist in plotting graphs and configuring the environment. We provide a `requirements.txt` file containing all necessary dependencies. To install the required Python libraries inside the `benchmark` directory each installed directory, run:

```
$ pip install -r requirements.txt
```

A.3.2 Basic Test

Once all repositories are set up, navigate to the `benchmark` directory within each repository and execute the command `fab localmal`. If all prerequisites are correctly installed, the user should see a message indicating "Running benchmark" along with the duration of the experiment. After the experiment completes, the `logs` directory will contain log files for each client and node. In the fault-free setting, a summary of statistics will also be displayed, providing key metrics such as throughput and latency. This output confirms that the system is functioning correctly

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *In the fault-free setting, our recovery protocol presents a slight decrease in throughput compared to the regular consensus protocol and presents a slight increase in the end-to-end latency.*

(C2): *In the excessive fault setting, our protocol returns the latency and throughput to its normal state after the recovery procedure is done.*

(C3): *The recovery time increases linearly with the logarithm of the number of blocks to retrieve.*

A.4.2 Experiments

Our results are all obtained from Cloud-based experiments on AWS. For the convenience of the artifact evaluation, we provide comprehensive instructions for users to run the experiments locally. The previous claims hold true in both local and cloud-based experiments. The last claim (C3) depends on the real-world network conditions, and the recovery time evaluated from local experiments. For completeness, we have included the instructions for running the experiments on AWS in the wiki of our Github repositories to facilitate replication of our results.

(E1): *[Fault free-setting compared to vanilla Hotstuff] [1 human-minute + 2 compute-hour]: For this experiment, we compare the throughput and latency of Hotstuff with our protocol implementation in a fault-free setting and the regular Hotstuff protocol.*

Preparation: *From the `fault-free-recovery` directory:*

```
$ cd benchmark
$ chmod +x get-results-fig1.sh
```

Execution: *From the `benchmark` directory, run:*

```
$ ./get-results-fig1.sh 10_000 120_000
7 5
```

This script executes our recovery protocol with 7 nodes, starting at an input rate of 10,000 transactions per second and incrementing by 10,000 transactions per second up to a maximum of 120,000 transactions per second. Each input rate is tested 5 times to ensure robust and reliable results.

Additionally, the script clones the vanilla Hotstuff codebase, runs it with the same parameters and copy the results to a `results` directory.

Run the same script for 30 nodes:

```
$ ./get-results-fig1.sh 10_000 120_000
30 5
```

Results: *Change the plot settings in* `fabfile.py` *to* `'nodes': [7, 30]`, `'faults': [0, 1]`, `'max_latency': [9_000]`. *Finally, run:*

```
$ fab plot
```

The graph should be in the `plots` *directory under the name of* `latency.pdf`. *The user will observe a graph similar to* ***Figure 2*** *with our results following the vanilla Hotstuff with a slight decrease in throughput and latency. This supports* ***C1***.

(E2): *[Normalized throughput and latency after recovery] [1 human-minute + 20 compute-minutes]:*

Preparation: *From the* `one-shadow-recovery` directory:

```
$ cd benchmark
```

We created a `setup-env.py` *script to assist in generating the required setup files. The script has multiple options, which can be checked with the* `-h` *flag. To reproduce our settings, run the command:*

```
$ python3 setup-env.py -n 31 -l 7 -c 2
-a 60
```

Then, change the `localmal` *settings in* `fabfile.py` *to* `'nodes': 31` *and* `'duration': 300`.

Make sure that the execution script has the correct permissions:

```
$ chmod +x run-fig3-and-fig4.sh
```

Execution: *From the* `benchmark` *directory, run:*

```
$ ./run-fig3-and-fig4.sh
```

Results: *The results for* ***Figures 3*** *and* ***4*** *can be found in the* `plots` *directory, under the filenames* `tps-recovery-node0.pdf`, `tps-recovery-node1.pdf`, *and* `latency-comparison.pdf`. *In the* `tps-recovery graphs`, *users will observe periodic spikes at fixed intervals. The intervals become shorter towards the end of the graph and after the second recovery procedure. In the latency comparison graph, a significant reduction in latency after the recovery is noticeable. This supports* ***C2***.

(E3): *[Linearity of the recovery time] [1 human-minute + 2 compute-hour]:*

Preparation: *From the* `one-shadow-recovery` directory:

```
$ cd benchmark
```

Execution: *We created a* `run-experiments.py` *script to assist in generating the results. Run the command:*

```
$ python3 run-experiments.py
```

Results: *Finally, run:*

```
$ python3 plot_fig5.py
```

The graph for ***Figure 5*** *should be on the* `plots` *directory with the name* `recovery-time.pdf`. *The graph should be similar to Figure 5 and the user should observe a linearly increasing recovery time. This supports* ***C3***.

A.5 Notes on Reusability

We provide the `setup-env.py` script to enhance the reusability of our artifact, enabling users to customize experiments beyond the paper's scope. This script allows users to configure the number of honest and malicious replicas, set network delays for local testing, and define a custom number of chains for equivocation scenarios. Additionally, as the benchmark vanilla HotStuff codebase, users can test multiple input rates for transactions and adjust transaction sizes to fit their specific use cases. The code is modularized, with easily identifiable components such as consensus, network, and store, making it easy for researchers and developers to modify, extend, or adapt it to their preferred tools and scenarios.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: NASS: Fuzzing All Native Android System Services with Interface Awareness and Coverage

Philipp Mao Marcel Busch Mathias Payer
EPFL, Lausanne, Switzerland

A Artifact Appendix

A.1 Abstract

The artifact contains NASS's source code which can be used to fuzz native system services on COTS devices. Futhermore the artifact contains scripts to run various aspects of the evaluation.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

NASS is designed to fuzz native system services on COTS devices. There is a danger of damaging the device, forcing a manual reset or making the phone unusable. For the artifact evaluation, all experiments running on real phones will be run on our testing devices exposed over ADB via port forwarding. NASS does not pose a risk to the host system it is run on.

A.2.2 How to access

The artifact is publicly available at `https://doi.org/10.5281/zenodo.15577630` and `https://github.com/HexHive/NASS`.

A.2.3 Hardware dependencies

NASS either runs against an arm64 COTS Android device or an emulator. To evaluate NASS against FANS, an arm64 server is required in order to run the emulator with kvm support. Our experiments on COTS devices rely on the following phones:

- Xiaomi Redmi Note 13
- Google Pixel 9
- Samsung S23
- OnePlus 12R
- Infinix X670

Note that for the artifact evaluation, we provide access to these phones remotly.

NASS's host component should run on any Linux system. We use an Ubuntu 24.04 docker container.

A.2.4 Software dependencies

We ship a dockerfile in our repository which sets up all the software dependencies.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To counter the hardware dependencies on an arm64 linux server and COTS Android devices, we provide access to an arm64 and x86 server. The access credentials are communicated in the `Artifact access` field on hotcrp.

The arm64 server is already setup with the emulator to reproduce the FANS evaluation (paragraphs with **FANS**). Access this server with `ssh -p 2222 ae@65.108.89.50` The x86 server has ADB access to our COTS testing devices, which we are forwarding from our local desktop. (paragraphs with **COTS**). Access this server with `ssh ae@65.108.89.50` See Figure 1 for the overview of the evaluation setup.

For both servers clone the NASS repository (if the folder is not present with)
`git clone https://github.com/HexHive/NASS`
Then navigate to the `./NASS` folder. Run the `./setup.sh` script. The script will setup NASS by building the docker. Then use `./run.sh` to spawn a shell in the NASS docker container.

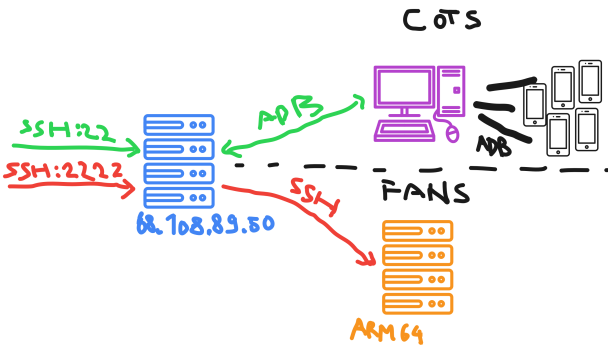


Figure 1: Overview of the evaluation setup.

A.3.2 Basic Test

All the steps here assume that you have logged in to the server, that `setup.sh` has finished successfully and the commands are inside of the NASS docker container.

COTS (ssh ae@65.108.89.50) Run the following command to list the devices connected to the server over ADB:

```
adb devices
```

The output should be the following:

```
List of devices attached
089092526K000893 device
47030DLAQ0012N device
RZCX312P76A device
a497c295 device
bai7gujvtchqeaus device
```

If this is not the case please let us know via hotcrp, it is possible the devices have shut down or the tunneling of ADB connections broke.

Assuming the five devices are present, run the following command which will fuzz a service on the Pixel 9 for 2 minutes.

```
./eval/cots/test_fuzz.sh
```

The output should contain the libfuzzer log lines akin to:

```
[LIBFUZZ] INFO: seed corpus: files: 1 min: 8b max: 8b total: 8b rss: 28Mb
[LIBFUZZ] #2 INITED ft: 42 corp: 1/8b exec/s: 0 rss: 28Mb
[LIBFUZZ] #4096 pulse ft: 42 corp: 1/8b lim: 4096 exec/s: 2048 rss: 29Mb
[LIBFUZZ] #8192 pulse ft: 42 corp: 1/8b lim: 4096 exec/s: 1638 rss: 29Mb
==========================
[ORC][47030DLAQ0012N] fuzzing iteration: 1 time: 62.078776836395264, crashes: 0, device borked: 0
[LIBFUZZ] #12486 NEW ft: 56 corp: 2/16b lim: 4096 exec/s: 1560 rss: 29Mb L: 8/8 MS: 1 ChangeCMd-
[LIBFUZZ] #16384 pulse ft: 56 corp: 2/16b lim: 4096 exec/s: 1489 rss: 29Mb
```

If this is displayed NASS was able to fuzz the `hardware.google.ril_ext.IRilExt/slot2` service on the Pixel 9.

FANS (ssh -p 2222 ae@65.108.89.50) Run the following command to fuzz a service on the arm64 emulator for 2 minutes.

```
./eval/fans/test_fuzz.sh
```

The output should contain the libfuzzer log akin to:

```
[ORC][emulator-5554] fuzzing iteration: 3 time: 24.192278385162354, crashes: 0, device borked: 0
[LIBFUZZ] #8192 pulse ft: 9 corp: 1/8b lim: 4096 exec/s: 372 rss: 28Mb
[LIBFUZZ] #9181 NEW ft: 70 corp: 2/16b lim: 4096 exec/s: 382 rss: 28Mb L: 8/8 MS: 1 ChangeCMd-
==========================
[ORC][emulator-5554] fuzzing iteration: 4 time: 30.4282705783844, crashes: 0, device borked: 0
[LIBFUZZ] #9245 NEW ft: 71 corp: 3/61b lim: 4096 exec/s: 385 rss: 28Mb L: 45/45 MS:
InsertEntry-
[LIBFUZZ] #9288 REDUCE ft: 71 corp: 3/40b lim: 4096 exec/s: 371 rss: 28Mb L: 24/24 MS: 1 InsertEntry-
[LIBFUZZ] #9295 REDUCE ft: 71 corp: 3/36b lim: 4096 exec/s: 371 rss: 28Mb L: 20/20 MS: 1 InsertEntry-
[LIBFUZZ] #9411 REDUCE ft: 71 corp: 3/24b lim: 4096 exec/s: 376 rss: 28Mb L: 8/8 MS: 1 DeleteEntry-
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Proprietary native services make up a large part of system services running on COTS devices (Table 1).

(C2): NASS finds bugs in proprietary services running on COTS devices (Table 5 and Table 6).

(C3): COTS services largely adhere to RPC design principles.

(C4): DGIE extracts accurate interface definitions from native services (Table 3).

(C5): NASS achieves parity with FANS (Figure 3, Table 4, Figure 4).

A.4.2 Experiments

All of the described commands assume that `setup.sh` been run on both servers and basic functionality test passed. We recommend to start a tmux session after logging in so longer-running commands are not terminated if the SSH connection drops. Simply run `tmux` after logging in over SSH. Furthermore all commands should be run inside of the NASS docker. To start the container use the `./run.sh` script. The following information including all the commands that need to be run can also be found in the `README.md` file.

COTS (ssh ae@65.108.89.50)

(E1): [Numbers of prop. native services, **C1**] [5 human-minutes + 15 compute-minutes]: Reproduce the numbers on proprietary native services on COTS devices from Table 1.
Preparation: Login to the COTS x86 server (ssh ae@65.108.89.50) and start the docker with `./run.sh`.
Execution: Run:
`./eval/cots/native-service-stats.sh`
Results: After the script has finished running, the numbers for each COTS device plus cumulative numbers are printed. These numbers should show that native services make up a large part of the system services running on COTS devices (**C1**) and be consistent with the numbers in Table 1. Note that the numbers do not match exactly since the firmware of the COTS devices has been updated since the evaluation was run for the paper.

(E2): [Fuzzing prop. native services, **C2**] [10 human-minutes + 3 compute-hours] Run NASS against proprietary native services on COTS devices.
Preparation: Login to the x86 server (ssh ae@65.108.89.50) and start the docker with `./run.sh`.
Execution: The provided script takes care of interface extraction, fuzzing and crash reproduction for a subset of native services. The script expects a device id. List these with `adb devices` then execute the run script with your chosen device id. Note it should be enough to execute the script once for a given device. For example, run:
`./eval/cots/run-fuzz.sh 47030DLAQ0012N`
to fuzz services on the Pixel 9.
Results: After the line `++EVAL++: analyzing crashes` the script will print the fuzzing output directories along with the reproduced crashes. Lines

starting with `========...` mark the fuzzed service. If no reproducible crashes were found `NO CRASHES REPRODUCED` is printed. If crashes were reproduced, the following line is printed `@@@@@@@@@@REPRODUCED AND DEDUPLICATED@@@@@@@@@@@@@@@@@@@@@@`, followed by the `output-path` and the abbreviated crash cause. After fuzzing the Pixel 9 we expect two crashes to have been found, an abort due to fortify and a null pointer dererefence. The `output-path` contains the results of the fuzzing campaigns, including the seed corpus, crashes etc. For reproduced crashes the full crashlog can be found in the `[output-folder]/reproduced/[crash-*]/crashlog.txt` folder.

The output of this script, ran across all proprietary native services on the five devices was used to produce Table 5 and Table 6. In the scope of the artifact evaluation we believe that demonstrating NASS' functionality on a subset of services is sufficent to reproduce this experiment.

(E3): [COTS service compliance, **C3**] [10 human-minutes + 1 compute-minutes] Double check the manual ground truth analysis on COTS service compliance with the three RPC design principles.

Preparation: Login to the x86 server (ssh ae@65.108.89.50) and start the docker with `./run.sh`.

Execution: The provided scripts aggregates the numbers from the manually analyzed ground truth (stored in `./eval/cots/dgie_eval.csv`). Run:

```
python3 ./eval/cots/dgie_cots.py
```

Results: The script should print the numbers from Table 7. The csv contains the result of the manual analysis of compliance of COTS services with RPC design principles. The columns Ab/St denote if the service is compliant with RPC design principles "Abstraction of IPC binding code" / "Standard Deserialization Routines". Filling these columns requires manually analyzing the services: The service binary is decompiled and the entry point function reverse engineered. We believe that reproducing the csv in this way is outside of the scope of this artifact evaluation. All the data needed to reproduce the numbers, the service binaries along with the offset to the entry point function, are part of the artifact stored at `eval/cots/dgie_cots_eval`.

FANS (ssh -p 2222 ae@65.108.89.50)

(E4): [DGIE interface extraction, **C3**] [15 human-minutes + 2 compute-hours]: The evaluation script first extracts the interface for the FANS evaluation services using DGIE and then collects the results along with the results from the manual ground truth and captured RPC requests to reproduce Table 3.

Preparation: Login to the arm64 server (ssh -p 2222 ae@65.108.89.50), start a tmux session, and then start the NASS docker with `./run.sh`.

Execution: Run: `./eval/fans/ground-truth.sh`

Results: After the script has finished, the part of Table 3 that is automatically generated is printed. Check consistency of the script output with the numbers in Table 3. The column `#NASS Extr. RPC. funcs.` was analyzed manually. To check that the interface extracted by DGIE matches the ground truth interface, you can manually inspect the extracted interface and compare it to the ground truth interface. Both these are stored in json format mapping command identifier to the sequence of deserializers.

To print the manually extracted ground truth for the `installd` service run

```
cat eval/fans/ground_truth/aarch64emu28/installd.json|jq
```

To print DGIE's automatically extracted interface run

```
cat targets/aarch64emu28/installd/preprocess/interface.json|jq
```

For the `installd` service the automatically extracted deserializers should match the manual ground truth ones. Note that the notitation of deserializers in the manual json does not exactly match the name from the preprocessing, see Table 1. The same step may be repeated if desired for the other FANS evaluation services.

Manual	DGIE
String16	STRING16UTF8
String16Vector	STRING16UTF8VECTOR
ByteVector	BYTEARRAY

Table 1: Mapping from manual interface deserializers to DGIE extracted.

(E5): [FANS fuzzing campaign **C4**] [10 human-minutes + 13 compute-hours] Run a fuzzing campaign on the FANS evaluation services with NASS, NASS (NI) and FANS.

Preparation: Login to the arm64 server (ssh -p 2222 ae@65.108.89.50), start a tmux session, and then start the NASS docker with `./run.sh`.

Execution: The provided script takes care of running a shortened (1 hour) fuzzing campaign and produces the coverage graphs. Run: `./eval/fans/run_eval.sh ae` to run the fuzzing campaign.

Results: After finishing the resulting coverage graph pdfs can be found in `./eval/fans/run_out/ae/`. Download these from the server with: `scp -r -P 2222 ae@65.108.89.50:/home/ae/NASS/eval/fans/run_out/ae_out .`

The y axis is the code coverage from 0% to 100%, the x-axis is the time. the resulting coverage graphs should look similar to the *first hour of the graphs in the paper*. We choose to run a shortened campaign due to the time required to run the full experiment (around 2 weeks).

A.5 Notes on Reusability

We provide documentation in the repository on how to adjust NASS to fuzz proprietary native system services on other devices.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Ariadne: Navigating through the Labyrinth of Data-Driven Customization Inconsistencies in Android

Parjanya Vyas
University of Waterloo
parjanya.vyas@uwaterloo.ca

Haseeb Ur Rehman Faheem
University of Waterloo
hfaheem@uwaterloo.ca

Yousra Aafer
University of Waterloo
yousra.aafer@uwaterloo.ca

N. Asokan
University of Waterloo
asokan@acm.org

A Artifact Appendix

A.1 Abstract

Vendor customization of the Android framework is known to introduce security concerns. One type of customization is data-driven, involving changes to access-controlled framework variables, which we call data holders. Analyzing the security of data-driven customization has not been explored in prior work because it faces several challenges as it requires modeling implicit access control (AC) relations among Java objects and their corresponding operation semantics. Existing Android AC inconsistency detection approaches struggle to discover data-driven AC inconsistencies.

We propose a novel approach, Ariadne, to address these challenges by (1) constructing an abstract representation, the AC dependency graph, to model AC relationships among framework data holders, and (2) using it to detect missing AC enforcement in data holders and their corresponding APIs. Using two AOSP and 11 custom ROMs, we show that Ariadne detects 30 unique data-driven AC inconsistencies which cannot be detected by existing approaches. Therefore Ariadne can offer more comprehensive protection by effectively complementing existing AC inconsistency detection approaches.

This artifact provides a proof-of-concept Java-based implementation of Ariadne. It takes a decompiled Android ROM as input and constructs an AC dependency graph. It then infers the expected AC enforcement for each API using a customized flooding algorithm. Each inference is assigned a *relevance weight* that reflects the tool's confidence in the inference. Finally, it outputs a list of APIs where the inferred AC is stronger than the existing enforcement, reporting only those exceeding a configurable relevance cutoff.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

No risk present for the evaluators, since the tool simply performs static analysis on the included dataset of ROMs.

A.2.2 How to access

The latest version of the artifact can be accessed at https://doi.org/10.5281/zenodo.15612788.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

The only dependency our tool has is Java 17.

A.2.5 Benchmarks

We provide all the decompiled ROMs used for evaluation in a directory called 'dataset'.

A.2.6 Interpreting the output

Each analysis of the individual ROMs performed by Ariadne creates a file called 'apis.csv' in the 'Output' directory. This file contains two columns:

- **API:** The name of the API.
- **Recommended Access Controls:** A list of AC checks recommended by Ariadne for the API.

Each access control recommendation in the second column includes the following fields:

`fromApi` The source API from which the access control requirement is originated.

`acType` The type of access control, e.g., `Permission`, `UID`, `AppOps`, etc.

`level` The normalized sensitivity level, such as `NONE`, `NORMAL`, `DANGEROUS`, `SYS_OR_SIG`.

`operator` The conditional operator used in the check (e.g., `EQ`, `GT`, `LT`).

`values` The hardcoded value compared against in the check (e.g., permission string, UID).

`access type` The type of access guarded, such as `GET`, `SET`, `MODIFY`, etc.

`weight` The relevance confidence score assigned to the recommendation.

Example: Consider the CSV entry representing `getExtensions` API with a recommended access control shown in Row 1 of Table 1. This entry indicates that `getExtensions` should be guarded by a UID-based system-level access control, inferred from the related `hasExtension` API, with a high confidence score of 0.85. The absence of this control is flagged as a data-driven inconsistency by Ariadne.

A.2.7 System used for experiments

All experiments were conducted on a MacBook Pro equipped with an Apple M3 Pro chip, 18 GB of RAM, and a 500 GB SSD, running macOS Sequoia.

A.3 Set-up

A.3.1 Installation

We provide the permanent artifact at the Zenodo repository mentioned above. For convenience, we also publish a docker image with all dependencies and setup.
To use the docker image:

1. Ensure docker is installed and available to use on your machine (https://docs.docker.com/get-docker/)
2. Run the following command to pull and run the docker image for Ariadne: `docker run -rm -pull=always -it parjanyavyas/ariadne:latest`

To build and run Ariadne setup from the Zenodo repository:

1. Install Java 17 and ensure it is available to use.
2. Download the artifact ZIP file and extract it.
3. Create two new sub-folders in the extracted folder called 'Input' and 'Output'.

A.3.2 Basic Test

Use the following command to run a basic test:

```
./gradlew startAnalysis
-Dpath="./Sample" -DoutPath="./Output"
-Dorg.gradle.java.home="<JAVA_HOME>"
-Duser.dir="."
```

After successful completion, verify that the "Output" directory contains the file 'apis.csv' with the content from Table 1.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Ariadne reports accuracy of 96.6% and 97.5% using AOSP Pixel 5 (v12) and AOSP Pixel 6 Pro (v13) respectively. Refer to Section 7.3 whose results are reported in column 6 of Table 6.
(C2): Ariadne has been used to uncover new inconsistencies in the included custom Android ROMs. Refer to Section 7.4 whose results are reported in column 5, Row 3-13 of Table 6.
(C3): Ariadne identifies APIs with new AC inconsistencies, and suggests appropriate AC checks, in the included custom Android ROMs.

A.4.2 Experiments

(E1): [30 human-minutes + 24-240 compute-hours (depending on underlying system used for the experiment) + 10GB disk]: Analyze AOSP ROMs to evaluate the accuracy of Ariadne by counting total number of APIs and inconsistencies in the two AOSP ROMs. For each of the two AOSP ROMs included in the dataset, perform the following:
Preparation: Create 'Input' and 'Output' folders if they do not already exist. Copy the decompiled AOSP ROM from dataset to 'Input' folder. Clear the 'Output' folder (if not already cleared).
Execution: Run the command as described in the README:

```
./gradlew startAnalysis
-Dpath="./Input" -DoutPath="./Output"
-Dorg.gradle.java.home="<JAVA_HOME>"
-Duser.dir="."
```

Results: The execution should produce an api csv file for the given ROM. This file can be interpreted as specified in Section A.2.6.

API Name	Access Control Annotation
android. service. securespaces. SecureSpacesService. getExtensions() Ljava/ util/ List;	ACAnnotation {acWithSrc = AccessControlSource{ fromApi = 'android. service. securespaces. SecureSpacesService. hasExtension(Ljava/ lang/ String;)Z', ac = ProgrammaticAccessControl{ acType=Uid, level=SYS_OR_SIG, operator=GT, values=[10000]}}, accessType=GET, weight=0.8573749999999999}
android. service. securespaces. SecureSpacesService. getUserRestrictions() Ljava/ util/ List;	ACAnnotation {acWithSrc = AccessControlSource{ fromApi = 'android. service. securespaces. SecureSpacesService. getDeviceOwnerUserRestrictions() Ljava/ util/ List;', ac = ProgrammaticAccessControl{acType=Uid, level=SYS_OR_SIG, operator=GT, values=[10000]}}, accessType=GET, weight=0.95}

Table 1: Access Control Annotations for SecureSpaces APIs

(E2): [2 human-hours + 120-1200 compute-hours (depending on underlying system used for the experiment) + 50GB disk]: Analyze custom ROMs to identify the inconsistencies detected by Ariadne. For each of the custom ROMs included in the dataset, perform the following:
Preparation: Create 'Input' and 'Output' folders if they do not already exist. Copy the decompiled custom ROM from dataset to 'Input' folder. Clear the 'Output' folder (if not already cleared).
Execution: Run the command as described in the README:

```
./gradlew startAnalysis
-Dpath="./Input" -DoutPath="./Output"
-Dorg.gradle.java.home="<JAVA_HOME>"
-Duser.dir="."
```

Results: The execution should produce an apis.csv file for the given ROM. This file can be interpreted as specified in Section A.2.6. Running the above command to analyze the Xiaomi Mix 2S ROM included in the dataset results in Ariadne identifying the two inconsistent APIs shown in Table 1, among other new APIs. This result confirms that Ariadne can identify these new access control inconsistencies.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Harness: Transparent and Lightweight Protection of Vehicle Control on Untrusted Android Automotive Operating System

Haochen Gong, Siyu Hong, Shenyi Yang, Rui Chang*,
Wenbo Shen, Ziqi Yuan, Chenyang Yu, and Yajin Zhou
Zhejiang University

A Artifact Appendix

A.1 Abstract

Harness is a lightweight framework that can transparently protect vehicle control on untrusted in-vehicle infotainment systems such as AAOS. Harness defines a minimal protection domain containing security-critical components related to car control. Leveraging hardware virtualization features, Harness isolates the domain, protecting components and sensitive interfaces within the domain from the untrusted OS. To demonstrate the feasibility of Harness, we implemented a prototype based on the Google Cuttlefish virtual platform and evaluated it on a Raspberry Pi 5 development board. This artifact provides the source code and instructions for building and running the prototype. We also provide pre-built images to ease the evaluation. For functionality, we present the main workflow of Harness via system logs and conduct an attack simulation. For reproducibility, we provide the benchmarks stated in the paper to evaluate the overhead of Harness and the scripts for processing the results.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Harness is a protection framework that does not pose any security risks. The evaluation needs to be conducted on a development board. Since we have modified the board's host kernel, which may cause some instability, we recommend backing up the existing data on the board in advance. Our attack simulation is conducted within a virtual machine and dose not affect the host. We provide an Internet-accessible board with a prepared environment to facilitate artifact evaluation. With this, reviewers do not need to worry about risks.

A.2.2 How to access

We host this artifact on Zenodo, and the DOI is 10.5281/zenodo.14723474, please refer to the latest version. During the AE period, reviewers can access our remote device.

A.2.3 Hardware dependencies

We evaluate Harness on a Raspberry Pi 5 development board with a 4-core Cortex-A76 64-bit 2.4 GHz Broadcom BCM2712 SoC and 8 GB RAM. Our artifact includes the modified AOSP components. Hence, an x86 machine is required to build the source code since AOSP can only be built on the platform. We recommend using Ubuntu 20.04 for the OS. In addition, building AOSP can be resource- and time-consuming. Please ensure the machine has over 300 GB of disk space and maintains a good network connection.

A.2.4 Software dependencies

Our prototype is developed based on Raspberry Pi kernel (rpi-6.6.y), AOSP (android-13.0.0_r41), and Android common kernel (common-android13-5.15). The Raspberry Pi kernel runs as the host kernel of the board. The host uses Google Cuttlefish (v0.9.29) to run AAOS within a virtual machine. To build the prototype, please ensure that the device has `git` and `repo` installed.

A.2.5 Benchmarks

All benchmarks we use for evaluation are provided in the artifact. Among them are two third-party benchmarks, LMBench 3.0 and Geekbench 5.5.1, which can be installed after the AAOS boot. The other benchmarks should be placed in the AOSP source tree and built into the system image.

A.3 Set-up

The Harness prototype can be divided into two parts: host and guest. The host part includes the modified Raspberry Pi kernel and a host kernel module containing the Harness Lowvisor (Section 4). The guest part includes the modified AOSP userspace framework and Android common kernel containing the Harness guest kernel module (Section 4). There are two ways to prepare the environment for evaluation. One

*The corresponding author.

is to build the prototype from scratch, and the other is to use pre-built images. If you choose the latter, please refer to the `Image Installation` section below directly. Reviewers using our remote device can download and follow the `README` from Zenodo to complete the evaluations.

A.3.1 Installation

Build from scratch.

1. *Build the host kernel.* Enter `src/harness_host/` and run `setup.sh`. The script will download, patch, and build the Raspberry Pi kernel. Please refer to the Raspberry Pi document [1] for more details.

2. *Build the guest Android common kernel.* Enter `src/harness_guest_kernel/` and run `setup.sh`. The script will automatically download, patch, and build the Android common kernel. The output images will be placed at `guest-kimage` in the directory. Please refer to document [2] for more details.

3. *Build the guest AOSP.* Enter `src/harness_aosp/` and run `setup.sh`. The script will automatically download, patch, and build the AOSP. This operation will take a few hours and the output images will be placed at `aosp-image` in the directory. Note that benchmarks in APK format are built during this time. Please refer to document[3] for more details.

4. *Build the Lowvisor.* Enter `src/harness_host/harness_host_kernel_module/` and run `make` to build the kernel module `harness.ko`.

Image Installation.

1. *Install the host kernel*. If using the prebuilt image, upload the `images/host-kernel/kernel_2712.img` to the `/boot/firmware` of the board. Otherwise, follow the Section *Cross-compile the kernel* in the document[1] to install the kernel on the boot media. Make sure that the device uses the `kernel_2712.img` as the kernel.

2. *Install guest images*. If using the prebuilt images, simply upload the `images` folder to the board and extract the AOSP archive files inside (a `.zip` and a `.tar.gz`) in need. Otherwise, upload the AOSP and Android common kernel images to the Raspberry Pi and extract the archive files. Place all the images in the same folder (the *working directory*).

Cuttlefish Installation. Upload the pre-built packages (`images/cuttlefish-prebuilt`) to the board and run the `install.sh`. Follow the Cuttlefish document[4] for details.

[1] https://www.raspberrypi.com/documentation/computers/linux_kernel.html
[2] https://source.android.com/docs/setup/build/building-kernels
[3] https://source.android.com/docs/setup/build/building
[4] https://source.android.com/docs/devices/cuttlefish/get-started

Lowvisor Installation. If using the prebuilt images, install the `.ko` file in the working directory using `insmod`. Otherwise, upload the `harness.ko` to the board and install it using `insmod`.

A.3.2 Basic Test

Enter the working directory and run:

```
HOME=$PWD ./bin/launch_cvd
-kernel_path=./guest-kimage/Image
-initramfs_path=./guest-kimage/initramfs.img
-daemon -cpus=4 -memory_mb=6144 -vhost_net=true
```

Run `./bin/adb shell` to connect to the device terminal. The guest log will be output to the `logcat` and `kernel.log` in `cuttlefish/instances/cvd-1/logs/`. Follow the Cuttlefish WebRTC document [5] to check if the virtual device is successfully booted and enters the home screen of AAOS.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Harness can transparently enclave userspace Andorid components (e.g., apps and services) without intrusive modification. This is proven by the experiment (E1), which demonstrates the functionality of Harness enclaves through system logs.*

(C2): *Harness can defeat potential attacks and protect the vehicle control chain of AAOS. This is proven by the attack simulation experiment (E2) described in Section 6.2 of the paper.*

(C3): *The performance and memory overhead of Harness are acceptable. This is proven by the experiment (E3) described in Section 6.3 of the paper, which covers a range of microbenchmarks (e.g., LMBench) and an application benchmark (Geekbench 5).*

A.4.2 Experiments

(E1): *[Basic Functionality] [30 human-minutes]: Harness can enclave userspace Andorid components (e.g., apps and services) without intrusive modification.*
How to: Combine system logs and process status to check the workflow and usage of Harness enclaves.
Preparation: Enable Lowvisor logging in the host kernel module source, then compile and install the module.
Execution: (1) During the AAOS boot, check the host system logs using the `dmesg` command. The Lowvisor will output the creation and execution status of enclaves. (2) After booting, connect to the virtual device using `adb`, using `ps -A` command to check process states. You will see the enclaved Zygote (`zygote64_enclaved`) running, and its child processes are also enclaved.

[5] https://source.android.com/docs/devices/cuttlefish/webrtc

(3) Using the `adb install` command to install the Geekbench (`benchmarks/Geekbench 5_5.5.1_APKPure.apk`) and run it. You will see the Geekbench process is forked from the enclaved Zygote.
Results: If the observed logs and process status meet expectations, the functionality of the Harness enclave can be validated. Since Geekbench's source code cannot be modified, its successful execution inside an enclave implies that no intrusive modifications are needed.

(E2): *[Security Features] [1 human-hours]: Harness can achieve our claimed security guarantee.*
How to: Run the attack simulation and check if the Lowvisor can detect malicious behaviors.
Preparation: (1) Memory-based attacks: Apply the patch `attack_sim/mattack_gkernel.patch` to the Android common kernel. Replace the host kernel with the `attack_sim/kernel_2712-att.img` and use the host kernel module in `attack_sim/`. Only use this image for memory-based attack simulation to avoid conflicts. Enter `attack_sim/sectests` and run `make`. Using `adb push` to upload the output binaries and `do_sectests.sh` to the virtual device.
(2) Malicious IPC: The `BinderBench` and `TestAppService` (in `aosp/device/harness/apps/`) work as a client-server pair using a protected interface. You can edit their `AndroidManifest.xml` to remove one of them from its enclave. Alternatively, you can use the prebuilt APKs (`BinderBench-nencl.apk` and `TestAppService-nencl.apk`) in the `benchmarks/`.
(3) Input injection: Uncomment the definition of the macro `HNS_ATTSIM_INPUT` in LibHarness (`aosp/framework/base/libs/harness/harness.cpp`). Enable the Lowvisor logging `HNS_EVTVF_DLOG`.
Execution: (1) Memory-based attacks: Run the script `do_sectests.sh` on the virtual device and check the log of the guest and host.
(2) Malicious IPC: Run the app `BinderBench` on the virtual device and click the `AutoBench` button. Check the `logcat`, and you will see the `Exception message` error, indicating that the transaction failed.
(3) Input injection: Open an enclaved app, click the virtual device screen 10 more times through WebRTC and check the host log.
Results: The system logs will output information about detected malicious behaviors.

(E3): *[Overhead of Harness] [1 human-hour + 1 compute-hour]: Harness incurs acceptable performance and memory overhead.*
How to: Run the benchmarks; Compare the memory usage when using Harness or not.
Preparation: (1) Compile and install the LMBench: We provide the binaries of LMBench and you can simply upload the `benchmarks/lmbench-3.0-a9-eval` folder to the virtual device using `adb push`. To build it from scratch, enter the folder and run `make CC=aarch64-linux-gnu-gcc OS=linux`.
(2) Install Geekbench: Using `adb install` to install `benchmarks/Geekbench 5_5.5.1_APKPure.apk`.
(3) Non-enclaved: To evaluate the impact on non-enclaved components, install the benchmarks in `benchmarks/non-enclaved` and repeat the tests.
Execution: (1) LMBench. Enter the `src` folder of LMBench and run `env OS="linux" ../scripts/results`. The results will be saved at `results/linux/localhost.0`.
(2) Binder latency. Launch the BinderBench app and click the `AutoBench` button. The result will be output into the `logcat`.
(3) Input responsiveness. Launch the EvalApp on the virtual device and use a clicker [6] to continuously click in the blue region labeled `TEST INPUT EVENTS`. Wait for 10 15 minutes and stop the clicker. The results will be recorded in the `logcat`.
(4) Car API latency. Launch the EvalApp on the virtual device and click the red region labeled `TEST CAR API`. Then, wait for the message "EvalCarAPI: Evaluation on Car API end" to appear in the `logcat`. The results will be recorded in the `logcat`.
(5) Geekbench. Launch the Geekbench on the virtual device and run `CPU BENCHMARK`.
(6) Startup time of apps. Run `cold_start.sh` and `hot_start.sh` (placed at `benchmarks/`) and redirect the output to a file.
(7) Memory usage. Run `eval_mem.py` (placed at `benchmarks/`) in the working directory. The script will output the average memory usage during a period.
Results: Repeat these tests in the baseline system, then refer to the README to process the obtained results using the script `eval.py`. The results should be close to those presented in the paper.

A.5 Notes on Reusability

Harness is not limited to protecting vehicle control but can also be applied to secure other Android components, similar to Android Virtualization Framework[7]. Please refer to Section 6.4 of our paper for detailed discussion.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

[6] https://github.com/InJeCTrL/ClickRun
[7] https://source.android.com/docs/core/virtualization



USENIX Security '25 Artifact Appendix: Scoop: Mitigation of Recapture Attacks on Provenance-Based Media Authentication

Yuxin (Myles) Liu
University of California, Irvine

Habiba Farrukh
University of California, Irvine

Ardalan Amiri Sani
University of California, Irvine

Sharad Agarwal
Microsoft

Gene Tsudik
University of California, Irvine

A Artifact Appendix

A.1 Abstract

Continuous advances in photo and video manipulation yield increasingly sophisticated deepfakes that greatly endanger societal perception of reality. Deepfake detection is an intuitive and natural research direction, which is unfortunately shaping up to be a never-ending arms race. An alternative promising direction is provenance assertion, which blends hardware-based secure camera design with the cryptographic means of authenticating the source of visual content and any post-processing (e.g., filters) applied to it.

This work starts by highlighting a very effective attack type, called a *recapture attack*, against all provenance-based techniques. In such an attack, the adversary displays fake content on some form of a screen (e.g., TV, projector, or computer screen) or surface (e.g., cardboard, canvas, or paper) and uses a provenance-asserting secure camera device to capture photos and videos of the displayed content.

We then introduce Scoop,[1] a systematic solution for mitigating recapture attacks. Scoop leverages state-of-the-art depth sensing technologies as well as learning-based depth estimation to detect misleading recaptures, i.e., a recaptured photo or video where the presence of a display medium is not visually identifiable.

We implement Scoop on both iOS and Android platforms (Apple iPhone 14 Pro and Samsung Galaxy S20 Plus), using their built-in depth sensors. To evaluate the effectiveness of Scoop, we construct a first-of-its-kind dataset consisting of 78 recapture attack scenarios. Our results show that Scoop achieves as high as $\approx 95\%$ accuracy on the iPhone and 74% accuracy on the Samsung phone.

A.2 Description & Requirements

There are four artifact components in our paper.

- **Viewer**: The Scoop Viewer is the main tool that we developed to detect the existence of misleading recaptures in photos/videos.
- **iOS App**: The iOS App is a mobile application that can be used to capture photos/videos that can be analyzed by the Scoop Viewer.
- **Android App**: The Android App is a mobile application that can be used to capture photos/videos that can be analyzed by the Scoop Viewer.
- **Dataset**: The dataset that can be used to evaluate systems such as Scoop.

A.2.1 Security, privacy, and ethical concerns

This artifact should not pose any threat to the evaluators machines' security, data privacy, or other ethical concerns. However, we recommend the use of VM when evaluating our artifact, which can help avoid any potential pollution on evaluators machines' environment.

A.2.2 How to access

All parts of our artifact can be found on our Zenodo page at https://doi.org/10.5281/zenodo.15611904. Each part contains a detailed README on its usage.

A.2.3 Hardware dependencies

Depending on different part of the artifact and different ways to evaluate it, various hardware might become necessary, such as a powerful GPU, a LiDAR-equipped iPhone, and an Android phone equipped with ToF sensor. As mentioned above, details can be found in each part's README.

A.2.4 Software dependencies

Depending on different part of the artifact and different ways to evaluate it, various software might become necessary, such as OpenCV, PCL, CUDA, and so on. As mentioned above, details can be found in each part's README.

[1] Scoop: **S**ecure **C**ontent **O**rigin from **O**ptical **P**roperties

A.2.5 Benchmarks

None. Although performance and storage overhead can be evaluated, they are not our main focus in the paper.

A.3 Set-up

There are three ways to test the Scoop system, we will only talk about one of them, which performs a quick overall test of our system, without the need of any special hardware. The other two ways to test our system can be found at subsection A.5 and A.6, where their detailed workflow can be found in the READMEs of our artifact.

We now provide a quick way to test out the Scoop system. Please download the Scoop Viewer (viewer.zip), which contains everything needed to conduct a quick test of Scoop.

A.3.1 Installation

The Scoop Viewer requires the following dependencies:

- C++ compiler (e.g., g++, clang++)
- CMake 3.5 or later
- OpenCV 4.5 or later
- PCL (Point Cloud Library) 1.12 or later
- Boost 1.75 or later
- Eigen 3.3 or later
- Python 3.9 or later

You may set up the dependencies yourself and modify the CMakeLists.txt file accordingly, or you can use our provided script to set up the environment automatically. Our script is tested on Ubuntu 24.04 LTS, but it should work on other Linux distributions (e.g., Fedora, Arch Linux) and MacOS (with Homebrew) as well. Assuming the viewer is now at a directory called `$VIEWER_DIR`, you can run the following commands to set up the environment:

```
cd $VIEWER_DIR
bash ./scripts/install_required_libraries.sh
```

A.3.2 Basic Test

After the installation is complete, you can run the following commands to build the Scoop Viewer:

```
cd $VIEWER_DIR
cmake .
make -j$(nproc)
```

After the build is complete, you can run the quick test with the following command:

```
cd $VIEWER_DIR
python3 ./scripts/eval.py sample_data/ 0000 9999
```

This command will run the Scoop Viewer on the sample data in the `sample_data` directory, which contains 8 sets of data provided (with 4 unique data points). Among the 4 data points, 1 is original and the rest 3 are recaptured photos (with 2 TVs and 1 projector).

A.4 Evaluation workflow

They are already mentioned above, and for detailed description or instructions, please refer to our READMEs.

A.4.1 Major Claims

(C1): A first-of-its-kind dataset for evaluating systems such as Scoop.
(C2): A functional iOS App to capture photos and videos that can be verified using Scoop's viewer.
(C3): A functional Android App to capture photos and videos that can be verified using Scoop's viewer.
(C4): A viewer that can analyze and detect visual content that can potentially mislead people into beleiving the existence of depth.

A.4.2 Experiments

The steps to conduct evaluation are already mentioned above, and for detailed description or instructions, please refer to our READMEs.

In terms of the expected duration of the evaluation, we expect an average time of 2 hours, with the majority time spent on setting up the environment. In terms of the outcomes of the experiments, we expect out system to successfully detect the misleading recapture scenes in the above quick test we provide. As for evaluation with self-produced data, we expect them to be detected properly as well in most cases, as long as the ground truth depth information is captured correctly. As for evaluation with our dataset, we expect the accuracy to match with our claims in the paper.

A.5 Test with Your Own Data

You can also test the Scoop Viewer with your own data. To do this, you can either use our iOS or Android app to capture photos/videos, or you can use your own camera to capture photos/videos. However, please make sure you follow the guidelines in the iOS/Android app repositories for correctly capturing the data or refer to the Scoop Viewer repository for the correct format of the data. For building iOS and Android apps, please refer to the respective repositories for instructions.

After you have extracted the data from your iOS/Android app, you need to generate perceived depth data for each

photo/video. Please refer to the Scoop Viewer repository for instructions on how to generate perceived depth data. You can pick any depth estimation model that you prefer, but we recommend using the ml-depth-pro model, which we included a script for using it in the Scoop Viewer repository. You may need to refer to the script to see how to generate the perceived depth data for your photos/videos and make sure the generated data is in the correct format so that the Scoop Viewer can analyze it. After you have generated the perceived depth data, you can run the Scoop Viewer on your data with the commands provided in the Scoop Viewer repository.

A.6 Test with the Dataset

You can also test the Scoop Viewer with our dataset. We provide full instructions on how to use the dataset in the Scoop Viewer repository. You can also refer to the Scoop dataset repository for more information about the dataset.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Chimera: Creating Digitally Signed Fake Photos by Fooling Image Recapture and Deepfake Detectors

Seongbin Park*, Alexander Vilesov*, Jinghui Zhang, Hossein Khalili,
Yuan Tian, Achuta Kadambi, Nader Sehatbakhsh
University of California, Los Angeles
*{parkseongbin,vilesov}@ucla.edu

A Artifact Appendix

A.1 Abstract

This paper introduces Chimera, an end-to-end attack strategy that crafts cryptographically signed fake images capable of deceiving both deepfake and image recapture detectors. Current adversarial and generative models fail to effectively deceive both detector types or lack generalization across different setups; Chimera addresses this gap by using a hardware-aware adversarial compensator to craft fake images that successfully bypass state-of-the-art detection mechanisms. This artifact demonstrates the effectiveness of Chimera fooling both types of detectors with a high success rate while having high visual quality (compared to the original real image). Our successful end-to-end attack on state-of-the-art detectors shows an urgent need for more robust detection strategies.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Since our attack does not target anything other than the target deepfake and recapture detectors (which are provided), there is no significant risk to the evaluators.

A.2.2 How to access

The complete artifact, including source code, pretrained models, and instructions, is available on Zenodo.

A.2.3 Hardware dependencies

None. However, using GPUs will speed up testing.

A.2.4 Software dependencies

The artifact should be evaluated in a conda environment. A complete list of software dependencies is provided in the `environment.yml` file, which includes PyTorch and other required dependencies. Additionally, the dataset and model weights used in the experiments are linked in the GitHub repository and must be downloaded.

A.2.5 Benchmarks

Pretrained models based on the datasets are provided in the conda environment. There is no need to download/access additional datasets.

A.3 Set-up

Follow these steps to prepare the environment for evaluation:

1. **Set up the Conda Environment:** Create and activate the conda environment using the provided `environment.yml`:

   ```
   conda env create -f environment.yml
   conda activate chimera
   ```

2. **Download the Dataset and Models:** The dataset is hosted on Zenodo. Please download it from the provided link (see the README). The dataset can be unzipped anywhere in the main directory; for the appropriate location of the models, refer to the next step.

3. **Configure Deepfake Detection:**

   - Edit `deepfake-detection/dataset_paths.py` to specify the correct paths to your dataset.
   - Move the following model weights into the `deepfake-detection/pretrained_weights` directory:
     - `fatformer_4class_ckpt.pth`
     - `fc_weights.pth`
     - `ViT-L-14.pt`

4. **Configure Recapture Detection:** Place the recapture detection model files into a directory of your choice. The paths to these models will be specified via command-line arguments during testing.

*These authors contributed equally to this work

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Chimera* can successfully bypass otherwise accurate **recaptured image** detectors. This is proven by the experiment (E1) described in Section 6.2 whose results are reported in Table 1.

(C2): *Chimera* can successfully bypass otherwise accurate **deepfake** detectors. This is proven by the experiment (E2) described in Section 6.3 whose results are reported in Table 4.

A.4.2 Experiments

(E1): *[Recapture Detection Experiment] [10 human-minutes + 10 compute-minutes]*

Preparation: No further preparation must be done from the steps outlined in A.3.

Execution: In the `recapture-detection` directory, execute the following command:

```
python main.py \
    --test \
    --test_path TEST_PATH \
    --config CONFIG \
    --test_raw_dirnames RAW_DIRNAMES \
    --test_recap_dirnames RECAP_DIRNAMES
```

where `TEST_PATH` is the path to the model and `CONFIG` is the path to the configuration file. Ensure that the configuration file matches the model being tested.
Specify the raw image directory with `RAW_DIRNAMES` and the recaptured image directory with `RECAP_DIRNAMES`. In the tar file provided, the raw images are in `stylegan2_orig`, benign recaptured images in `recap_mac`, and adversarial recaptured images in `attack_mac`.

Results: Upon completion, the script should output the accuracy of the models for the specified datasets.

(E2): *[Deepfake Detection Experiment] [2 human-minutes + 5 compute-minutes]*

Preparation: No further preparation must be done from the steps outlined in A.3.

Execution: In the `deepfake-detection` directory, run the test script:

```
./test.sh
```

This script runs all deepfake detection tests sequentially.

Results: The script saves the evaluation results (including detection metrics and success rate analyses) in the `deepfake-detection/results` directory.

A.5 Notes on Reusability

Though our method is generalizable to any hardware setup, the adversarial generative model must be trained specifically for each setup.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Tracking You from a Thousand Miles Away! Turning a Bluetooth Device into an Apple AirTag Without Root Privileges

Junming Chen
George Mason University

Xiaoyue Ma
George Mason University

Lannan Luo
George Mason University

Qiang Zeng*
George Mason University

A Artifact Appendix

The nRootTag artifact demonstrates a novel attack that transforms computers into trackable "AirTags" without requiring root privileges by exploiting Apple's Find My network. The attack exploits a vulnerability in Apple's Find My network, which Apple has acknowledged. We also assisted Apple to reproduce the attack. Apple recently released patches in iOS 18.2, visionOS 2.2, iPadOS 17.7.3, 18.2, watchOS 11.2, tvOS 18.2, macOS Ventura 13.7.2, Sonoma 14.7.2, Sequoia 15.2 to fix the vulnerability. However, the attack remains effective as long as unpatched iPhones or Apple Watches are in the proximity of the computer running our trojan.

A.1 Abstract

The artifact includes all components needed to reproduce the attack: a command & control server setup script for managing tracked devices, a custom database for efficient key storage, a GPU-accelerated key search program, and proof-of-concept trojans for Linux, Windows, and Android systems. The artifact provides comprehensive documentation, screen recordings, and three pre-computed rainbow tables to facilitate the reproduction of the key findings.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our research involves experimentation with a live third-party system, i.e., Apple Find My network. The Trojan for Linux is designed to write a bytestream to */dev/rfkill* to enable Bluetooth functionality. It attempts to connect to a specified address, defaulting to localhost, to transmit the Bluetooth adapter address and request a public key. Subsequently, it advertises the public key, thereby facilitating the tracking of the test device. The Trojans for Android and Windows are configured to attempt Bluetooth activation with or without user consent and to send Bluetooth advertisements for the exchange of addresses.

Evaluators shall exercise extra caution when handling the location reports, as possession of the corresponding private key permits decryption of the report and revelation of the location. It is advised that evaluators refrain from extracting private keys from artifacts and from broadcasting lost messages at private locations (e.g., residential addresses); instead, a new key should be generated using Key Seeker. Evaluators should not share the private key or location report publicly.

Table 1: Hardware configurations used for each component

Component	Hardware Requirement
C&C Server	Alienware Aurora R5 Desktop
Key Seeker	Alienware Aurora R5 Desktop, NVIDIA RTX 4090
Trojan - Linux	Alienware Aurora R5 Desktop
Trojan - Android	Google Pixel 6, Pixel 4
Trojan - Windows	Gigabyte Z690, Bluetooth Adapter RTL8761B

A.2.2 How to access

We have released the artifact on Zenodo[1] to comply with the Open Science Policy. The artifact includes a README, instructional videos, Docker scripts for setup, reproduction codes for lost message search and advertisement, online key search guidelines, and three precomputed rainbow tables. We also host a corresponding GitHub page[2] for issuing potential bug fixes and accepting pull requests.

A.2.3 Hardware dependencies

Table 1 details the hardware configuration in our experiment.

*Corresponding author.

[1] https://doi.org/10.5281/zenodo.14728530
[2] https://nroottag.github.io/

Table 2: Software configurations used for each component

Component	Software Requirement
C&C Server	Ubuntu 22.04
Key Seeker	Ubuntu 22.04
Trojan - Linux	Ubuntu 22.04
Trojan - Android	Android 13, Android 11
Trojan - Windows	Windows 11

A.2.4 Software dependencies

Table 2 shows the software setup, which is the same as the software environment used in the demonstration videos.

If evaluators have further requirements for confirming the submission and decryption of location reports, evaluators may refer to online projects like *Chapoly1305/FindMy*.[3] We highlight that this process is beyond the scope of our paper and we do not vouch for any of those projects. To comply with the single-blind review policy, we may not be able to decrypt location reports for evaluators.

A.2.5 Benchmarks

None, our work does not compare with other datasets.

A.3 Set-up

In the README file, we provide detailed steps to replicate our work. In addition to the README file, the demonstration videos show the essential steps and results related to the key findings presented in the paper. We recommend that evaluators refer to these materials for a better replication experience.

A.3.1 Installation

The C&C Server is written in Python and uses several libraries. We provide two methods to set up the environment; either method should work.

```
# Assume current directory is nRootTag
# Method A, install from requirements
  python3 -m venv venv
  source venv/bin/activate
  pip install -r requirements.txt

# OR, Method B, use our packed venv
  tar xvf venv.tgz
  source venv/bin/activate

# Lastly, Start Server
  python3 cnc_server.py
```

[3] https://github.com/Chapoly1305/FindMy

A.3.2 Basic Test

After the server is launched, evaluators can access the webpage at `http://localhost:7898/docs` to review the available APIs and their usage. The webpage presents each API with a description and usage examples. Evaluators can also use the "Try it out" feature for each API and follow the inline descriptions to test its functionality.

A.4 Evaluation workflow

A.4.1 Major Claims

We made the following claims in our main paper.

(C1): Key Seeker achieves a key generation and searching speed approximate of 8.23 billion KPS using a single RTX 4090. It is proven by experiment **E1**, as described in Section 6.4 of our paper.

(C2): The finder devices in Apple's Find My network do not distinguish between different types of Bluetooth addresses while processing lost messages. It is proven by experiment **E2**, described in Section 6.4 of our paper.

(C3): As described in Section 6.4 of our paper, the Trojan is capable of turning on Bluetooth on devices running Linux, Windows 11, and Android versions prior to 13, as proven by experiment **E3**. The detailed information of devices is listed in Table 5 and Table 6 of the main paper.

A.4.2 Experiments

(E1): [Key Seeker Benchmark] [0.5 human-hour]
Preparation: In this experiment, the Key Seeker is evaluated for correspondence to **C1**.

Execution: Navigate to nRootTag/executables and run the following commands. In the given example code, **-t 0** disables searching with the CPU, **-gpu** enables searching with the GPU, **-o /dev/null** discards all outputs.

```
chmod +x ./Seeker_CUDA_12
./Seeker_CUDA_12 -t 0 -gpu -o /dev/null -p
```

Results: Employing an RTX 4090, the anticipated rate is approximately 8.23 Gkey/s. Variability in the outcomes may occur based on the GPU's state and potential concurrent usage by other applications. Similar experiments can be conducted with alternative GPU models. A line commencing as follows denotes the average speed of GPU processing.

```
[Time 896.3s][7.51 Gkey/s][GPU 8.23 Gkey/s]
```

(E2): [Lost message advertisement] [1 human-hour + 22 compute-hours]
Preparation: To validate claim **C2**, the experiment entails retrieving the Bluetooth adapter's address from a Linux-based system. Following this, the procedure for

initiating the private key search is demonstrated. For this purpose, one might employ an RTX 4090 to search for a key pair associated with the Bluetooth address. Achieving a precise match, where all 48 bits of the address correspond to the 48 most significant bits (MSBs), the probability of identifying a matching key is estimated to be 90% within 22 hours, as delineated by Equation 1 in our study. Evaluators might opt to utilize cloud-based GPU platforms and leverage multiple GPU units to decrease the computational time required, as discussed in Section 6.1 of the paper.

Execution: Evaluators shall take note of the device Bluetooth address for later comparison.

```
hcitool dev
```

Evaluators shall start the server by using the given script and keep the script running. The script will open a port and listen on port 7898.

```
python3 cnc_server.py
```

The evaluators should deploy the Trojan onto the testing apparatus. Upon the Trojan's execution, a search task is initiated on the server. The extracted Bluetooth address is then output by the Trojan. The console will exhibit the phrase "Advertisement Registered" once a matching key is identified. Following the retrieval of a public key from the C&C server, the Trojan will commence advertising activities.

```
# Launch Trojan
  chmod +x ./Trojans/trojan_linux.py
  bash -c "Trojans/trojan_linux.py"
# Launch Key Seeker
  python3 seeker_standalone.py
```

Results: Evaluators should verify the consistency of the given address across the specified address, the Web API for /search-task, and the Trojan interface. Evaluators should check the public and private keys utilizing the API /review-key. Trojan advertisements can be detected using Bluetooth scanning tools such as *nRF Connect*.[4] Evaluators should look for advertisements with a company ID of 0x004C from the specified public address. To confirm advertisements handled by the locator, evaluators can query Apple through external software like *Chapoly1305/FindMy*. When Apple devices, such as iPhones with the Find My feature activated, are present, evaluators should be able to retrieve location reports promptly.

(E3): [Bluetooth Stealth Enable] [2 human-hours]

Preparation: Evaluators shall install with the operating system as described in **C3**, in accordance with Table 5 and Table 6 of our main paper.

Execution: Evaluators are required to disable the Bluetooth function within the system configurations for each operating system. On Linux, the script *trojan_linux.py* shall be executed. For Android systems, evaluators need to install the application *nRootTag-Android.apk* and initiate the nRootTag Trojan program. Meanwhile, on Windows 11, the file *nRootTagWinRT.exe* should be launched by the evaluators.

Results: On all three platforms (i.e., Linux, Android, Windows), evaluators should observe the Bluetooth becoming active after launching Trojan, irrespective of the status of the address for advertising purposes.

A.5 Notes on Reusability

The artifact can be employed in future research related to the Find My network. Extensive in-line comments are included to facilitate the reuse of the platform's components, particularly Key Seeker, which is the core element of the research and can be utilized for cryptography-related studies. A separate document containing notes is provided in the *executables/nRootTag-Seeker.zip/Developer.md* file. This design document offers a detailed plan for implementing cryptographic algorithms across various elliptic curves, with a specific focus on the transition from SECP256K1 to SECP224R1.

The document provides a comprehensive explanation of the GPU-CPU architecture, memory management, and mathematical optimizations, which can be invaluable for researchers and developers working on similar cryptographic implementations. Furthermore, the document thoroughly discusses performance considerations, such as thread management, batch processing, and buffer sizing, offering practical advice for scaling these systems. The explicit mention of potential challenges, including the need to maintain Y coordinates and the consequences of modifying bit-sizes, helps future developers avoid common pitfalls.

The inclusion of Python code examples and detailed explanations of mathematical operations enhances the accessibility and implementability of the concepts presented. Finally, the documentation of specific modifications necessary for different curves, along with the rationale behind each change, provides a roadmap for adapting the system to other elliptic curves, making it a valuable resource for future cryptographic brute-force search and research in GPU-accelerated cryptography.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

[4] https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp



USENIX Security '25 Artifact Appendix: Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis

Youngjoon Kim
Korea University

Sunguk Shin
Korea University

Hyoungshick Kim*
Sungkyunkwan University

Jiwon Yoon*
Korea University

A Artifact Appendix

A.1 Abstract

Paper: Research on automated vulnerability repair often requires extensive program analysis and expert input, making it challenging to deploy in practice. We propose SAN2PATCH, a system that generates patches using only sanitizer logs and source code, eliminating the need for costly program analysis or manual intervention. SAN2PATCH employs multi-stage reasoning with Large Language Models (LLMs) to decompose the patching process into four distinct tasks: vulnerability comprehension, fault localization, fix strategy formulation, and patch generation. Through tree-structured prompting and rigorous validation, SAN2PATCH can generate diverse, functionally-correct patches. Evaluations on the VulnLoc dataset show that SAN2PATCH successfully patches 79.5% of vulnerabilities, surpassing state-of-the-art tools like ExtractFix (43%) and VulnFix (51%) by significant margins. On our newly curated SAN2VULN dataset of 27 new vulnerabilities from various open-source projects, SAN2PATCH achieves a 63% success rate, demonstrating its effectiveness on modern security flaws. Notably, SAN2PATCH excels at patching complex memory-related vulnerabilities, successfully fixing 81.8% of buffer overflows while preserving program functionality. This high performance, combined with minimal deployment requirements and elimination of manual steps, makes SAN2PATCH a practical solution for real-world vulnerability remediation.

Artifact: This artifact includes not only the source code of SAN2PATCH proposed in the original paper, but also the VulnLoc benchmark with added functional tests and our newly constructed SAN2VULN benchmark. All artifacts are dockerized for easy setup, and we provide bash scripts to reproduce all experiments presented in Section 4 of the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

SAN2PATCH is an automated program repair tool that generates patches for vulnerabilities. As a result, the tool itself does not pose any security, privacy, or ethical concerns. To prevent any issues that may arise during the reproduction process, we provide a Docker environment for additional isolation.

A.2.2 How to access

Our artifact is permanently accessible through a DOI link[1], GitHub repositories[2,3], and pre-built Docker images available on Docker Hub[4,5]. Any download method is acceptable; however, we recommend downloading the Docker images directly from Docker Hub for the most seamless setup experience.

A.2.3 Hardware dependencies

There are no specific constraints as long as the environment supports Python 3.10 or Docker. However, we recommend a CPU with at least 4 cores, 16 GB or more of RAM, and at least 100 GB of available disk space for reliable performance. Note that the hardware used for the experiments in the original paper comprised an Intel Xeon Gold 5218 CPU with 32 cores (2.30 GHz) and 64 GB of RAM.

A.2.4 Software dependencies

If the artifact is executed directly on the host system, a Python 3.10 environment and Poetry version 2.1.1 or higher for Python environment management are required. When using the Docker image, Docker version 28 or higher is recommended.

A.2.5 Benchmarks

This artifact provides the benchmark used in the original paper as `san2patch-benchmark`. The benchmark consists of an extended version of VulnLoc with added functional tests and our newly constructed SAN2VULN benchmark containing recent vulnerabilities. To avoid dependency conflicts, the benchmark must be executed in an independent Docker container. When the `san2patch` Docker image is used, Docker-in-Docker must be employed to run the `san2patch-benchmark` Docker image within the `san2patch` container.

*Corresponding authors

[1] https://doi.org/10.5281/zenodo.15614211
[2] https://github.com/acorn421/san2patch
[3] https://github.com/acorn421/san2patch-benchmark
[4] https://hub.docker.com/r/acorn421/san2patch/
[5] https://hub.docker.com/r/acorn421/san2patch-benchmark/

A.3 Set-up

This section assumes the use of our pre-built `san2patch` Docker image. For instructions on running the artifact in a host environment, please refer to the `README.md` file within the `san2patch-main` repository.

A.3.1 Installation

We recommend using the provided Docker images for easier environment setup. Pulling and running the `san2patch` Docker image automatically handles all necessary configurations, including launching the required `san2patch-benchmark` Docker image.

```
docker pull acorn421/san2patch
docker run -it --privileged --name san2patch acorn421/san2patch
```

Running the image automatically starts the Docker server and pulls the `san2patch-benchmark` Docker image. Pulling the image may require a noticeable amount of time (approximately 30 minutes). Once the image has been successfully launched, set the LLM API keys directly in `/app/.env` as shown below. For more information on configuring the environment, refer to the `README.md` in `san2patch-main`.

```
# Required: LLM API Keys (at least one)
OPENAI_API_KEY=your_openai_api_key_here
ANTHROPIC_API_KEY=your_anthropic_api_key_here
GOOGLE_API_KEY=your_google_api_key_here
```

A.3.2 Basic Test

```
python ./run.py Final run-patch --vuln-ids "CVE-2016-1839" --
      model "gpt-4o" --experiment-name "test"
```

To verify whether the test ran successfully, check the following directory:

```
cd ./benchmarks/final/final-test/gen_diff_test
```

A.4 Evaluation workflow

After launching the Docker image and completing the setup, a bash shell will be available. In this shell, scripts prepared in the `/app/experiments` directory can be executed to reproduce the results of each RQ in the Evaluation section of the original paper.

A.4.1 Major Claims

(C1): SAN2PATCH *achieves a 79.5% patch success rate (31/39 vulnerabilities) on the VulnLoc benchmark, significantly outperforming existing approaches. This is demonstrated by the evaluation in Section 4.1.*

(C2): SAN2PATCH *effectively patches recent and unseen vulnerabilities in the* SAN2VULN *benchmark at low cost ($0.48 per attempt) and within practical timeframes (under 9 minutes per patch), as shown by the results in Section 4.2.*

(C3): *The performance of* SAN2PATCH *improves with more advanced LLMs, as its effectiveness relies on LLM capabilities. This is demonstrated in Section 4.3.*

(C4): *The ToT-based patch generation methods, No Context and* SAN2PATCH*, yield a higher proportion of functionally correct patches, as reported in Section 4.4.*

(C5): SAN2PATCH *effectively patches various types of vulnerabilities, as demonstrated by the experiments in Section 4.5.*

(C6): *An ablation study shows that removing stages such as Vulnerability Comprehension and How-To-Fix from* SAN2PATCH *reduces performance, indicating that all stages are essential. This is proven by the experiment in Appendix A.*

A.4.2 Experiments

How to Run Each experiment can be executed by running the corresponding script file. If execution does not proceed as expected, ensure that the `san2patch-benchmark` Docker container is running properly inside the `san2patch` Docker-in-Docker container.

Results The results of each experiment are saved in:

```
/app/benchmarks/final/final-test/gen_diff_{experiment_name}
```

where `experiment_name` is a unique identifier for each experiment, as specified in the script files.

Key result files can be found under each ID of the vulnerability directory:

- `res.txt` – Summary of all attempts, including status codes.
- `*_success.diff` – The final patch that passed all automated tests (vulnerability and functionality tests)
- `*_success.artifact` – All intermediate information generated during the successful patching process.

To check the automated validation results of the experiment, refer to `res.txt`. To simply count the number of successful patches, use the following command:

```
cd /app/benchmarks/final/final-test/gen_diff_{experiment_name}
ls ./**/res.txt | xargs -I {} bash -c "echo '===== {} ====';
     cat {}; echo;" | grep success | wc -l
```

To conduct manual validation, examine each `*_success.diff` file associated with vulnerabilities

that were successful in automated validation. Detailed criteria for manual validation can be found in the supplementary material[6].

Please refer to the `README.md` in `san2patch-main` for further details on additional result files.

(E1): *VulnLoc Benchmark Evaluation [4 scripts]:*
This experiment evaluates the vulnerability patching capability of SAN2PATCH on the VulnLoc benchmark, supporting claims **(C1)**, **(C4)**, and **(C5)**.
Estimated Time: Each script is executed with 4-core parallel processing for 6 hours, for a total of 6×4 hours.
Estimated Cost: Less than $50 per script with GPT-4o, requiring less than $\$50 \times 4$ in total.
Execution: Run the provided scripts for each method as follows:

```
bash ./experiments/rq1/tot_vulnloc.sh
bash ./experiments/rq1/cot_vulnloc.sh
bash ./experiments/rq1/no_context_vulnloc.sh
bash ./experiments/rq1/zeroshot_vulnloc.sh
```

Results: The results are stored in the corresponding output directories:

```
ls ./benchmarks/final/final-test/
     gen_diff_usenix_tot_vulnloc
ls ./benchmarks/final/final-test/
     gen_diff_usenix_cot_vulnloc
ls ./benchmarks/final/final-test/
     gen_diff_usenix_no_context_vulnloc
ls ./benchmarks/final/final-test/
     gen_diff_usenix_zeroshot_vulnloc
```

(E2): SAN2VULN *Benchmark Evaluation [1 script]:*
This experiment evaluates the vulnerability patching capability of SAN2PATCH on the SAN2VULN benchmark, supporting claim **(C2)**.
Estimated Time: The script is executed with 4-core parallel processing for 4 hours, for a total of 4 hours.
Estimated Cost: Less than $40 in total with GPT-4o.
Execution: Run the provided script as follows:

```
bash ./experiments/rq2/usenix_tot_san2vuln.sh
```

Results: The results are stored in the following output directory:

```
ls ./benchmarks/final/final-test/usenix_tot_san2vuln
```

(E3): *LLM Model Comparison [5 scripts]:*
This experiment compares the vulnerability patching capability of SAN2PATCH using different LLM models, supporting claim **(C3)**.
Estimated Time: Each script is executed with 4-core parallel processing for 6 hours, for a total of 6×5 hours per script.
Estimated Cost: Less than $50 per script with GPT-4o, requiring less than $\$50 \times 5$ in total.
Execution: Run the provided scripts for each model as follows:

```
bash ./experiments/rq3/gpt_4o_mini.sh
bash ./experiments/rq3/gpt_35.sh
bash ./experiments/rq3/sonnet_35.sh
bash ./experiments/rq3/gemini_15_pro.sh
bash ./experiments/rq3/gemini_15_flash.sh
```

Results: The results are stored in the corresponding output directories:

```
ls ./benchmarks/final/final-test/
     gen_diff_usenix_gpt_4o_mini
ls ./benchmarks/final/final-test/gen_diff_usenix_gpt_35
ls ./benchmarks/final/final-test/gen_diff_usenix_sonnet_35
ls ./benchmarks/final/final-test/
     gen_diff_usenix_gemini_15_pro
ls ./benchmarks/final/final-test/
     gen_diff_usenix_gemini_15_flash
```

(E4): *Ablation Study [2 scripts]:*
This experiment investigates the impact of removing certain reasoning stages from SAN2PATCH, supporting claim **(C6)**.
Estimated Time: Each script is executed with 4-core parallel processing for 6 hours, for a total of 2×6 hours.
Estimated Cost: Less than $50 per script with GPT-4o, requiring less than $\$50 \times 2$ in total.
Execution: Run the provided scripts as follows:

```
bash ./experiments/ablation/no_comprehend_k5.sh
bash ./experiments/ablation/no_howtofix_k5.sh
```

Results: The results are stored in the following output directories:

```
ls ./benchmarks/final/final-test/
     gen_diff_usenix_no_comprehend
ls ./benchmarks/final/final-test/
     gen_diff_usenix_no_howtofix
```

A.5 Notes on Reusability

Occasionally, the Python script may not terminate automatically even after all experiments have finished. This does not indicate a failure; the experiments are still considered successfully completed.

When running experiments in parallel, it is possible to exceed the API rate limit. It is recommended to monitor the logs to ensure that rate limiting does not occur during execution.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

[6] https://doi.org/10.5281/zenodo.15654492





USENIX Security '25 Artifact Appendix: APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching

Yu Nong[1], Haoran Yang[2], Long Cheng[3], Hongxin Hu[1], and Haipeng Cai[1*]

[1]*University at Buffalo*, [2]*Washington State University*, [3]*Clemson University*

[1]{yunong,hongxinh,haipengc}@buffalo.edu, [2]haoran.yang2@wsu.edu, [3]lcheng2@clemson.edu

A Artifact Appendix

A.1 Abstract

APPATCH is an automated LLM-based patching system that elicits LLMs to effectively reason about vulnerable code behaviors with vulnerability semantics reasoning and adaptive prompting. We provide access to the APPATCH datasets, results, source code, and a functional Docker image described in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our research is focused on the development and evaluation of methods for patching software vulnerabilities on the source code level, which does not have any risk for evaluators when executing the artifact with regards to the machines' security, data privacy, or other ethical concerns.

A.2.2 How to access

We have made the APPATCH datasets, results, and source code publicly available on Zenodo: https://zenodo.org/records/14741018 with a functional Docker image on Docker Hub: https://hub.docker.com/repository/docker/g2ecb/appatch-demo/.

A.2.3 Hardware dependencies

Our artifact can run on machines with at least 20GB spare disk space and 16GB CPU memory.

A.2.4 Software dependencies

Our functional APPATCH image requires Docker to set up and execute.

A.2.5 Benchmarks

Considering the high cost of the commercial LLMs, we only provide 16 interprocedural samples for the functional badge evaluation. For the complete datasets and results, please refer to appatch.zip uploaded on Zenodo.

A.3 Set-up

A.3.1 Installation

We have uploaded the functional Docker image to Docker hub. To pull and install the image, simply run:

```
$ docker run -it -d --name appatch-demo
  g2ecb/appatch-demo bash
```

After installing the image, enter the container with:

```
$ docker exec -it appatch-demo bash
```

After entering the Docker container, please first provide the keys in the api_keys.json file.

```
$ cd ~
$ vi api_keys.json
```

A.3.2 Basic Test

In the home directory (`/root/`) of the container, run the automatic script provided for the whole pipeline:

```
$ source run_all.sh
```

If the API keys are set correctly, you will see the script run without reporting model related errors.

[*]Haipeng Cai is the corresponding author.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): APPATCH is able to generate vulnerability patches with its key components: (1) Semantics-Aware Scoping (Slicing), (2) Vulnerability Semantics Reasoning (Root Cause Analysis), (3) Dynamic Adaptive Patch Generation, and (4) Multi-Faceted Patch Validation. We demonstrate the claim through the demo Docker image in Experiment (E1).

A.4.2 Experiments

(E1) Patch Generation *(1 human-minute + 20 compute-minutes)*:

How to: Run the code in the execution section below.

Preparation: Set up the API keys described in Section A.3.1. Then, go to the home directory of the container.

Execution: Run the automatic script provided for the whole pipeline:

```
$ source run_all.sh
```

Results: After running the pipeline, the generated slices, root cause analysis, patches, and validation results are stored in `interprocedural_sample_slices`, `root_cause_analysis`, `generated_patches`, and `generated_patches_<model-name>_valid` folders in `/root/appatch/`. By opening any generated files in `root_cause_analysis` and `generated_patches`, you can see results similar to Figure 5 and Figure 7 in the paper.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: RangeSanitizer: Detecting Memory Errors with Efficient Range Checks

Floris Gorter
Vrije Universiteit Amsterdam
f.c.gorter@vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

A Artifact Appendix

A.1 Abstract

In this artifact we provide instructions to reproduce our main results. This artifact concerns our tool named RangeSanitizer (RSan), which detects spatial and temporal memory errors in C/C++ programs. Our results show that RSan can successfully detect memory errors, and does so with higher runtime performance than traditional redzone-based bug sanitizers. We have validated the artifact using a system containing an Intel i9-13900K CPU and running Ubuntu 22.04 with a stock Linux v5.15 kernel. Our source code is available at: https://github.com/vusec/rangesanitizer

A.2 Description & Requirements

The experimental setup of RSan (with implicit pointer tagging by default) concerns an x86 system running Ubuntu. Using explicit pointer tagging requires hardware support. We require the evaluators to obtain the SPEC CPU benchmarking suite themselves, since we cannot distribute the licensed software.

A.2.1 Security, privacy, and ethical concerns

As a bug sanitizer, RSan poses no risks to the security and privacy of the target machine, and has no ethical implications.

A.2.2 How to access

The files for the artifact evaluation are currently available at: https://github.com/vusec/rangesanitizer/releases/tag/ae and https://zenodo.org/records/14701524

A.2.3 Hardware dependencies

By default, we evaluate the implicit pointer tagging design that supports any (legacy) x86 architecture. If desired, RSan's explicit pointer tagging can be used, which requires CPU hardware support: either Arm TBI (Armv8-a and onwards) or Intel LAM (Lunar/Arrow Lake). In order to support some of the benchmarks, 32 GB of RAM is recommended.

A.2.4 Software dependencies

While RSan does not have very specific software requirements, some packages from the Ubuntu package manager are required to be installed to build RSan's dependencies (e.g., the LLVM project). These are described in the Set-up section. We evaluated RSan on Ubuntu 22.04.

A.2.5 Benchmarks

For this artifact we benchmark using the SPEC CPU2006 benchmarking suite, and the Juliet test suite.

A.3 Set-up

We recommend using a bare-metal desktop system with (at least) 32 GB of RAM, running Ubuntu 22.04, glibc 2.35, and a stock v5.15 Linux kernel.

A.3.1 Installation

1. Obtain the artifact source from the `ae` release:

```
git clone \
https://github.com/vusec/rangesanitizer.git \
 --recurse-submodules --branch ae
cd rangesanitizer
```

2. Install some standard dependencies:

```
sudo apt install ninja-build cmake gcc-9 \
autoconf2.69 bison build-essential flex \
texinfo libtool zlib1g-dev unzip

pip3 install psutil terminaltables
```

3. Configure the RSan environment by editing the `env.sh` file and modifying the `RSAN_TOP` variable to reflect the working directory of the system, and then run:

```
source env.sh
```

4. We recommend changing the CPU scaling governor to `performance` for stability in benchmarks (requires sudo):

```
echo "performance" | sudo tee \
 /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

5. Install the RSan infrastructure by running:

```
./install-all.sh
```

NOTE: installing LLVM can take up a lot of RAM when using multiple cores. If the compilation process crashes (OOM), modify the `ninja -j <cores>` parameter inside `install-all.sh` to use fewer cores.

A.3.2 Basic Test

To test the basic functionality of RSan, we provide two test programs in the `examples` directory. To compile and run these tests, execute:

```
cd examples
./test-implicit.sh
```

See README.md in the artifact for the exact expected output format (returned addresses may vary).

A.4 Evaluation workflow

In order to run the Juliet test suite and SPEC CPU along with its benchmarks we make use of a public infrastructure under the `infra` directory. The infra also makes sure the SPEC binaries are pinned to core 0. Make sure that the necessary python packages are installed (see the Installation section).

A.4.1 Major Claims

(C1): RSan can detect spatial and temporal memory errors bounded by its security guarantees (as described in Section 7.1). This is proven by experiment E1.

(C2): RSan provides high performance in terms of runtime overhead (see Section 7.3). This is proven by experiment E2.

A.4.2 Experiments

(E1): [30 compute-minutes]: Confirming memory error detection.

How to: The Juliet test suite contains buggy programs for which RSan can detect the bugs at runtime.

Preparation: Ensure that the `env.sh` and `install.sh` scripts have been executed successfully. Note that we run Juliet with optimization flag -O0, because standard optimizations (e.g., -O2) hide bugs in the test cases. Feel free to also test Juliet with AddressSanitizer (ASan): change the target of setup.py from `rsan-impl_O0` to `asan_O0` in the command below.

Execution: Execute the following command, which builds and runs the relevant Juliet categories (see Table 1):

```
python3 setup.py run juliet rsan-impl_O0 \
--build --parallel=proc \
--parallelmax=$(nproc) \
--cwe 121 122 124 126 127 415 416
```

Results: The exact expected output can be found in the README.md file in the artifact. All tests are expected to pass.

(E2): [5 compute-hours]: Confirming runtime performance.

How to: Run the SPEC CPU2006 benchmarking suite instrumented by RSan and ASan, and observe the performance overhead compares to a baseline (non-instrumented) run.

Preparation: SPEC CPU2006 needs to be available on the system and the `RSAN_SPEC2006` variable in `env.sh` needs to point to the directory where it is installed. For the artifact evaluators, if they cannot obtain SPEC CPU2006, we can provide access to a machine ready to run SPEC. Ensure that the `env.sh` and `install.sh` scripts have been executed successfully (`env.sh` needs to be reloaded after modifying its content).

Execution: Execute the following command, which builds and executes SPEC CPU2006 for three runs: the baseline, one with ASan, and one with RSan, which in total takes multiple hours:

```
python3 setup.py run spec2006 baseline_O2 \
rsan-impl_O2 asan_O2 --build \
 --parallel=proc --parallelmax=1
```

Results: The exact expected output can be found in the README.md file in the artifact. Note that there will be a corresponding output folder in the `results` directory which can be inspected. To obtain a summary of the results from the SPEC CPU2006 runs, again make use of the `setup.py` script. Execute the following command:

```
python3 setup.py report spec2006 \
results/last --field runtime:median \
maxrss:median
```

The output of this command can then be used to calculate the runtime and memory overheads for each individual binary, as well as for the geomean. As reported in Figure 9: if ran on an x86 machine with an i9-13900K CPU, the expected runtime overhead for RSan is 51%, and 95% for ASan. See the README.md file in the artifact for a complete example output and the corresponding geomean overhead calculation.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Attacker Control and Bug Prioritization

Guilhem Lacombe
Université Paris-Saclay, *CEA*, *List*, *France*
guilhem.lacombe.97@gmail.com

Sébastien Bardin
Université Paris-Saclay, *CEA*, *List*, *France*
sebastien.bardin@cea.fr

A Artifact Appendix

A.1 Abstract

As bug-finding methods improve, bug-fixing capabilities are exceeded, resulting in an accumulation of potential vulnerabilities. There is thus a need for efficient and precise bug prioritization based on exploitability. In this work, we explore the notion of control of an attacker over a vulnerability's parameters, which is an often overlooked factor of exploitability. We show that taint as well as straightforward qualitative and quantitative notions of control are not enough to effectively differentiate vulnerabilities. Instead, we propose to focus analysis on feasible value sets, which we call *domains of control*, in order to better take into account threat models and expert insight. Our new *Shrink and Split* algorithm efficiently extracts domains of control from path constraints obtained with symbolic execution and renders them in an easily processed, human-readable form. This in turn allows to automatically compute more complex control metrics, such as *weighted Quantitative Control*, which factors in the varying threat levels of different values. Experiments show that our method is both efficient and precise. In particular, it is the only one able to distinguish between vulnerabilities such as *cve-2019-14192* and *cve-2022-30552*, while revealing a mistake in the human evaluation of *cve-2022-30790*. The high degree of automation of our tool also brings us closer to a fully-automated evaluation pipeline.

This artifact contains the necessary material to reproduce all experimental results from the paper, including the source code of our tool Colorstreams, our benchmarks and scripts to run experiments and analyze results. In addition, we provide tutorials and documentation for Colorstreams.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact does not pose any threat to the security and privacy of evaluators.

A.2.2 How to access

The artifact is openly available on Zenodo: https://doi.org/10.5281/zenodo.14699098.

A.2.3 Hardware dependencies

There are no specific hardware requirements but we recommand having a CPU with at least 16 cores.

A.2.4 Software dependencies

Any OS that can run docker is suitable. For reproducing the experiments without the docker image, any Linux distro compatible with the Nix package manager should be suitable, although the artifact was only tested on Ubuntu.

Here is a list of Colorstreams' main software dependencies, which are automatically handled through Nix and thus **do not need to be installed manually**:

Binsec: symbolic execution
Intel PIN: tracing
Z3: SMT solver
Bitwuzla: SMT solver
Q3B: SMT solver
popcon: model counting, includes d4
ganak: model counting
approxmc: model counting
gdb: debugging

Note that this list does not include sub-dependencies and the (many) benchmark dependencies. For more details, check out the artifact's Nix files (*nix/sources.json* in particular).

A.2.5 Benchmarks

The experiments require two benchmarks. The first was assembled by us and the second is based on the MAGMA fuzzing benchmark (https://hexhive.epfl.ch/magma/). The artifact contains both of them with documentation; no manual setup is required.

A.3 Set-up

A.3.1 Installation

With docker: Download *docker_image.tar.gz* from Zenodo, then run *docker image load < docker_image.tar.gz*.

To start a container, run *docker run -it –rm colorstreams:usenix25*. We recommend setting up a shared volume to easily transfer files from the container, such as figures. To do so, add *-v <local_dir>:<mount_point>* to the container startup command (paths must be absolute). For example, the full command could be *docker run -it –rm -v /home/aaa/bbb:/share colorstreams:usenix25*. Alternatively, *docker cp* can also be used, although it is less convenient.

Without docker: Given the amount of dependencies, we only provide compilation through Nix. This should also help to future-proof the building process. To install Nix, follow the instructions at https://nixos.org/download. Then, download *colorstreams.tar.gz* from Zenodo, decompress it and open a shell in the artifact's root directory. To build and install *colorstreams*, run *make* and *sudo make install*. To build the benchmark programs, run *make xps_build*.

A.3.2 Basic Test

To test the artifact, run *make test* ($\sim$ 15 minutes). Then, run *make test_stats* and *make test_results*. You should obtain results similar to Listings 1 and 2 respectively.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our Shrink and Split algorithm is more precise than the state-of-the-art (Newsome et al.) and even competitive against model counters.

(C2): Our Shrink and Split algorithm is faster than the state-of-the-art (Newsome et al.) and more robust that model counters.

(C3): Our prioritization approach using weighted quantitative control is more precise than other scoring methods.

(C4): Our method can be used in realistic end-to-end prioritization scenarios.

A.4.2 Experiments

If you are using the docker image, please refer to Section A.3.1 to start a container. Otherwise, open a shell inside the artifact's root directory.

(E1): Mainline experiments (10 human-minutes + 10 compute-hours): Analyses of both benchmarks for reproducing the results from the paper, excluding appendices.
Execution: run *make xps_less*
Results: run *make figs_less SAVEFIG=<directory>*. Figures will be saved to the directory specified with SAVEFIG, which can be your shared directory if you are using docker and set up one.
(C1): fig1.pdf (Figure 1), fig3.pdf (Figure 2)
(C2): tab4.txt (Listing 3), speedups.txt (Listing 4)
(C3): tab5_oob.txt (Listing 5), tab5_cfh.txt (Listing 6)
(C4): fig5.pdf (Figure 3)

(E2): Full experiments (10 human-minutes + 20 compute-hour or 10 compute-hours if (E1) was run first): Analyses of both benchmarks for reproducing all results from the paper, including appendices.
Execution: run *make xps_full*
Results: run *make figs_full SAVEFIG=<directory>*. Figures will be saved to the directory specified with SAVEFIG, which can be your shared directory if you are using docker and set up one.

You can find more details about the experiments in the artifact's readmes.

A.5 Notes on Reusability

The artifact includes tutorials and API documentation for Colorstreams, as well as instructions for using and expanding the benchmarks. Please refer to the readmes (in *colorstreams.tar.gz*).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

```
> make test_stats
Executed instructions and analysis runtime (s) for each bug:

Bug          Analysis                 Symbolic Instr.   Total Instr.   SE Runtime   Total Runtime
-----------  -----------------------  ----------------  -------------  -----------  -------------
can          Symbolic Policy <1>                    2           2709    0.0055747        14.6759
can2         Symbolic Policy <1>                    3           2830    0.010097         17.4844
ccopy        Symbolic Policy <1>                    6             27    0.00972962        7.04594
cfi          Symbolic Policy <1>                   12           2249    0.0351627         7.89045
cfi2         Symbolic Policy <1>                    5           2218    0.0258272         7.94325
copy         Symbolic Policy <1>                    2             22    0.00235724       12.9396
div          Symbolic Policy <1>                    3             23    0.00308895       16.2794
grub         Symbolic Policy <1>                   84           3868    0.200363        107.172
impflow      Symbolic Policy <1>                    3             40    0.00736499        6.63708
koobe        Symbolic Policy <1>                   12           4075    0.0235684        26.1484
mcopy        Symbolic Policy <1>                    3             23    0.00285673        6.39718
minesweeper1 Symbolic Policy <1>                  278          68911   16.3686          96.9821
minesweeper2 Symbolic Policy <1>                  254         107110   13.5707         107.404
mixdup       Symbolic Policy <1>                   13             33    0.0125771        63.9717
motex1       Symbolic Policy <1>                   18             78    0.0429707        30.1603
motex2       Symbolic Policy <1>                   15           1848    0.0337915        15.7375
mul          Symbolic Policy <1>                    3             23    0.00335526       35.8923
popcnt       Symbolic Policy <1>                   39             59    0.045434          7.2513
spray        Symbolic Policy <1>                    8           2224    0.0165174        24.4337
spray2       Symbolic Policy <1>                   16           2248    3.29333          80.0886
spray3       Symbolic Policy <1>                   20             89    3.73511          58.8148
sub          Symbolic Policy <1>                    3             24    0.00478458        5.81582
sum          Symbolic Policy <1>                    4             32    0.00477386       43.1932
uafubi       Symbolic Policy <1>                   11           2851    0.738554        178.45
uafubi2      Symbolic Policy <1>                    2           2843    0.00483179       15.5785

Total SE runtime:  38.19642400741577
Total overall runtime:  978.8097233772278
```

Listing 1: Result of *make test_stats*

```
> make test_results
Results compared to ground truth (d4 or ganak, when available):

Algorithm                 True (TP/TN)   Approx (<100%/>100%)   False (FP/FN)   Timeout   Unknown   Total Runtime
------------------------  -------------  ---------------------  --------------  --------  --------  -------------
Taint                     24/2           –                      4/0             –         –         0s
WeakControl               24/6           –                      0/0             –         –         0s
StrongControl             7/23           –                      0/0             –         –         1s
d4                        26             –                      –               4         –         42s
ganak                     26             –                      –               4         –         40s
approxmc                  23             1/0                    –               4         2         59s
Newsome                   10             14/2                   –               0         4         8m54s
S&S (100 max splits)      23             1/2                    –               0         4         47s
S&SFB (100 max splits)    25             0/1                    –               0         4         47s
```

Listing 2: Result of *make test_results*

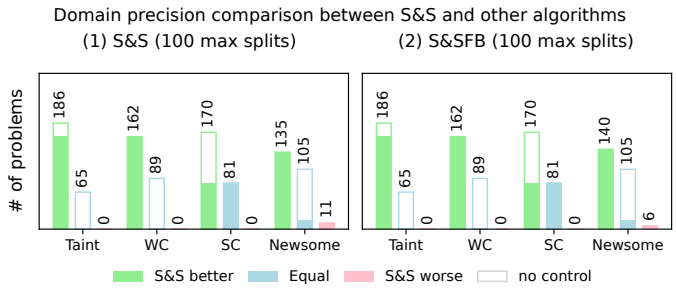


Figure 1: fig1.pdf

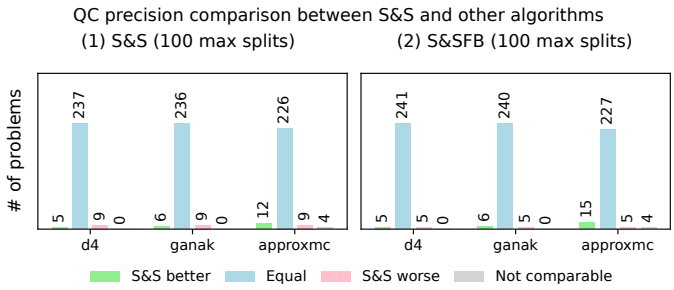


Figure 2: fig3.pdf

Results compared to ground truth (d4 or ganak, when available):

Algorithm	True (TP/TN)	Approx (<100%/>100%)	False (FP/FN)	Timeout	Unknown	Total Runtime
Taint	162/65	–	24/0	–	–	0s
WeakControl	162/89	–	0/0	–	–	3s
StrongControl	81/170	–	0/0	–	–	46s
d4	246	–	–	5	–	35m35s
ganak	245	–	–	6	–	32m49s
approxmc	228	18/0	–	1	4	15m08s
Newsome	105	137/4	–	1	5	2h08m48s
S&S (10 max splits)	235	6/5	–	0	5	10m56s
S&S (50 max splits)	237	4/5	–	0	5	11m12s
S&S (100 max splits)	237	4/5	–	0	5	11m52s
S&S (500 max splits)	237	4/5	–	0	5	15m19s
S&S (1000 max splits)	237	4/5	–	0	5	19m33s
S&SFB (10 max splits)	238	7/1	–	0	5	10m26s
S&SFB (50 max splits)	240	5/1	–	0	5	10m45s
S&SFB (100 max splits)	241	4/1	–	0	5	10m55s
S&SFB (500 max splits)	241	4/1	–	0	5	13m05s
S&SFB (1000 max splits)	241	4/1	–	0	5	15m27s

Listing 3: tab4.txt

```
Runtime averages (5% trimmed):
    Newsome: 30.8 (22.2)
    S&S (10 max splits): 2.62 (0.505)
    S&S (50 max splits): 2.68 (0.611)
    S&S (100 max splits): 2.84 (0.64)
    S&S (500 max splits): 3.67 (0.637)
    S&S (1000 max splits): 4.67 (0.622)
    S&SFB (10 max splits): 2.5 (0.491)
    S&SFB (50 max splits): 2.57 (0.564)
    S&SFB (100 max splits): 2.61 (0.558)
    S&SFB (500 max splits): 3.13 (0.574)
    S&SFB (1000 max splits): 3.7 (0.569)

Speedup averages (5% trimmed):
    S&S (10 max splits): 79.4 (71.5)
    S&S (50 max splits): 80 (71.2)
    S&S (100 max splits): 80.9 (72.9)
    S&S (500 max splits): 83.1 (73.2)
    S&S (1000 max splits): 82.8 (73.4)
    S&SFB (10 max splits): 87.2 (77)
    S&SFB (50 max splits): 84.7 (75.4)
    S&SFB (100 max splits): 88.9 (77.5)
    S&SFB (500 max splits): 84.9 (74.9)
    S&SFB (1000 max splits): 84.8 (75.3)
```

Listing 4: speedups.txt

Basic scores based on counts and domains:

Bug	Sink	Size	QC	wQC
motex2	of_wsize <1>	64	0.0832	0.0837
motex1	of_wsize <1>	64	0.0832	0
heartbleed	payload_size <11>	32	0.5	0.5
cve-2023-37837	read_off <0>	16	1	1
cve-2022-30790_2	of_woff2 <5>	64	0.249	0.037
cve-2022-30790	of_woff <7>	64	0.0248	0.0066
cve-2022-30790	of_wsize <5>	64	0.219	8.59e-08
cve-2022-30552	of_woff <3>	64	0.0248	0.0066
cve-2022-30552	of_wsize <1>	64	0.219	8.59e-08
cve-2021-3246	of_size <1>	64	0.275	0.221
cve-2019-14202	of_wsize <1>	32	0.499	0.499
cve-2019-14192	of_wsize <1>	32	0.499	0.499

Listing 5: tab5_oob.txt

Note: to obtain the scores from the paper, multiply the scores in the table by the bit size of the sink. For vulnerabilities with two parameters, add both scores together.

CFH capability scores:

Bug	ID	Type	QC Score	wQC Score
cve-2020-14393.cfh	0	Jump	0.996	1
cve-2023-43338.cfh	0	Call	1.53e-05	1
cve-2021-26567.cfh	0	Ret	0.969	0
cve-2024-41881.cfh	0	Ret	0.939	0
cve-2024-43700.cfh	0	Ret	0.776	0

Listing 6: tab5_cfh.txt

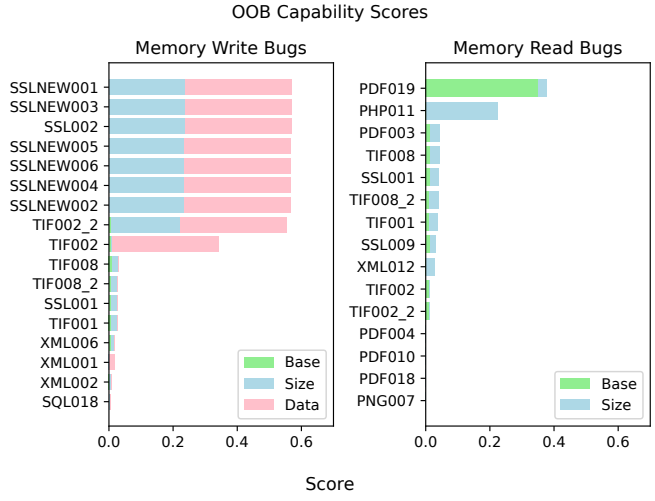


Figure 3: fig5.pdf





USENIX Security '25 Artifact Appendix: SafeSpeech: Robust and Universal Voice Protection Against Malicious Speech Synthesis

Zhisheng Zhang[1], Derui Wang[2] ✉, Qianyi Yang[1], Pengyang Huang[1], Junhan Pu[1], Yuxin Cao[3], Kai Ye[4], Jie Hao[1] ✉, and Yixian Yang[1]

[1]*Beijing University of Posts and Telecommunications* [2]*CSIRO's Data61*

[3]*National University of Singapore* [4]*The University of Hong Kong*

A Artifact Appendix

A.1 Abstract

In our artifact, we provide the source code with a detailed description of SafeSpeech in the public communities. We have implemented the algorithmic design of SafeSpeech and open-sourced our code at https://github.com/wxzyd123/SafeSpeech on GitHub and https://zenodo.org/records/15118529 on Zenodo with the final version. In this artifact, we introduce step-by-step instructions on reproducing the experimental results of the main experiments in the paper, including dependence description, environment installation, data preparation, and running commands with expected outputs.

A.2 Description & Requirements

We provide minimum hardware and software requirements for reproducing our experiments.

Hardware Requirements: CPU: a device with 100GB; GPU: NVIDIA A800 with 80GB memory or other GPU devices with the same memory.

Software Requirements: Operating system: Linux (Ubuntu 20.04); Python 3.8; PyTorch 2.0.0; CUDA 12.4. The required Python libraries are provided in `requirements.txt`.

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns in this artifact.

A.2.2 How to access

The source code for SafeSpeech is available on GitHub. You can download it directly from the repository page or use the Git tool to clone it. The repository includes the SafeSpeech implementation and the LibriTTS dataset.

✉ Corresponding authors: `derek.wang@data61.csiro.au`, `haojie@bupt.edu.cn`.

A.2.3 Hardware dependencies

We recommend a device equipped with the Ubuntu operating system version 20.04 for hardware configuration. In our tests, the CPU memory is 100 GB. We recommend using a device with more than 80GB for the GPU, such as the NVIDIA A800 described in our paper.

A.2.4 Software dependencies

For software configuration, our experiments are conducted in an environment with Python 3.8, PyTorch 2.0.0, and CUDA 12.4. The SafeSpeech implementation is built on the PyTorch framework and includes additional Python libraries, which are specified in the `requirements.txt` file.

A.2.5 Benchmarks

In this paper, we evaluate the effectiveness, transferability, and robustness of SafeSpeech primarily on speakers from the LibriTTS [2] dataset who are most similar to the pre-trained speakers on BERT-VITS2 [1] model for perturbation generation and fine-tuning. The original data is provided in the `data/LibriTTS` folder within the source code, requiring no further modifications.

A.3 Set-up

A.3.1 Installation

Our experiments are conducted in a Conda environment. First, install the CUDA toolkit (version 11.8 or higher) to ensure proper dependency setup. Then, you can install the Conda environment from the website https://www.anaconda.com/download.

For environment creation, we provide the corresponding instructions in the "Setup" section of the `README`.

(1): `conda create -name safespeech python=3.8`: This command creates a conda environment named "safespeech" with Python version 3.8.

(2): `conda activate safespeech`: This switches to the "safespeech" conda environment.

(3): `pip install -r requirements.txt`: This command installs the necessary Python dependencies required to run the experiment.

(4): `sudo apt install ffmpeg`: This command installs the "ffmpeg" software.

A.3.2 Basic Test

To check whether PyTorch has been successfully installed, you can try adding the following lines to a .py file with "import torch, touchaudio" and "print(torch.cuda.is_available())". Then, run the script. If the command line output is "True", it means the basic test has passed.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): SafeSpeech achieves the MCD value of 14.771 after fine-tuning on LibriTTS dataset and the BERT-VITS2 model, reflecting the higher protective performance than baselines in Table 1;
(C2): The evaluation values of WER and MCD are 99.610% and 0.204 which are better than baselines.

A.4.2 Experiments

(E1): *[Pre-trained Models][1 human-minute + 10 compute-minutes]* For downloading the pre-trained models of the BERT-VITS2, you can run this command:
Command: python download_models.py;
Output: The downloaded checkpoints.

(E2): *[Protection][1 human-minute + 25 compute-minutes]* Since LibriTTS is already available in the GitHub repository, you can start the experimental evaluation once the pre-trained models are prepared. The evaluation process is mainly divided into protection, fine-tuning, and evaluation.
First, you need to generate the BERT file for each audio by DeBERTa model:
Command: python bert_gen.py –dataset LibriTTS –mode clean;
Output: BERT files.
The generated BERT files can be found in the `data/LibriTTS/protected/clean` folder. After obtaining BERT files, the protected audio can be obtained by:
Command: python protect.py –dataset LibriTTS –batch-size 27 –mode SPEC;
Output: protective perturbations.

(E3): *[Saving][1 human-minute + 2 compute-minutes]* The generated perturbations have been saved to `checkpoints/LibriTTS/noises` folder; then you can add the perturbations to the original audio to get protected data:
Command: python save_audio.py –mode SPEC –batch-size 27;
Output: protected audio.
The saved audio files can be found in the folder `data/LibriTTS/protected/SPEC`.

(E4): *[Fine-tuning][1 human-minute + 20 compute-minutes]* Following the aforementioned commands, the dataset has been protected. You can utilize these audio files for fine-tuning. Also, the BERT files need to be generated:
Command: python bert_gen.py –dataset LibriTTS –mode SPEC;
Output: BERT files.
The generated BERT files of protected audio can be found at `data/LibriTTS/protected/SPEC`. Then, you can fine-tune BERT-VITS2 by:
Command: python train.py –mode SPEC –batch-size 64;
Output: checkpoint of fine-tuning.
The checkpoint of the generator after fine-tuning has been saved to `checkpoints/LibriTTS`.

(E5): *[Evaluation][1 human-minute + 10 compute-minutes]* The last step of the workflow is an evaluation after fine-tuning by this command:
Command: python evaluate.py –mode SPEC;
Output: the synthesized results of three metrics.
The synthesized audio can be found in the folder `evaluation/LibriTTS/SPEC` and the output information is:
Mode SPEC, MCD: XXX
Mode SPEC: GT WER is 0.000000, Syn WER is XXX
Mode SPEC on LibriTTS, SIM XXX, ASR XXX.
These three lines represent the evaluation metrics MCD, WER (*i.e., Syn WER*), and SIM, respectively. Your results may not be the same as ours (MCD 14.771, WER 99.610, and SIM 0.204), but they should be higher than baselines, *i.e.*, the values in Table 1, D_1 dataset, under the BERT-VITS2 model, excluding the SPEC row.

Additionally, to facilitate quick validation of SafeSpeech, we provide a Colab version via Jupyter Notebook that includes environment setup and experimental replication steps.

A.5 Notes on Reusability

In this artifact, we provide detailed instructions on how to reproduce SafeSpeech on a new device and environment. All the pre-trained models and datasets are open-source, and we have introduced how to download and protect them by SafeSpeech.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] Bert-vits2. https://github.com/fishaudio/Bert-VITS2, 2024.

[2] H. Zen, V. Dang, R. Clark, Y. Zhang, R. J. Weiss, Y. Jia, Z. Chen, and Y. Wu. Libritts: A corpus derived from librispeech for text-to-speech. *arXiv*, 2019.



USENIX Security '25 Artifact Appendix: SoK: Automated TTP Extraction from CTI Reports – Are We There Yet?

Marvin Büchel[†,1], Tommaso Paladini[†,2,6], Stefano Longari[2], Michele Carminati[2], Stefano Zanero[2], Hodaya Binyamini[4], Gal Engelberg[4], Dan Klein[4], Giancarlo Guizzardi[3], Marco Caselli[5], Andrea Continella[3], Maarten van Steen[3], Andreas Peter[1], and Thijs van Ede[3]

[1]Carl von Ossietzky Universität Oldenburg, [2]Politecnico di Milano, [3]University of Twente, [4]Accenture Labs, [5]Siemens AG, [6]NEC Laboratories Europe

A Artifact Appendix

A.1 Abstract

This Artifact Appendix covers the experiments and the resulting main conclusions from our Empirical Analysis part of the SoK paper. As already mentioned in the paper, it is not about the absolute values, but the relative comparisons among each other, especially in the approaches itself. All experiments are for evidence only and are not intended to present a new high-performance prototype. For this reason, we aim to evaluate the relative trends in the approaches. Our artifact is structured in three main *sub-projects*, each requiring their own setup. Each sub-project (`NER`, `classification`, `generation`) can be found in a folder inside the main directory of the artifact, and contain the code related to the corresponding systematization experimental section. `NER`: Contains the code related to the NER approaches (Sections 2.2, 4); `classification`: Contains the code related to the Classification approaches (Sections 2.3, 5); `generation`: Contains the code related to the Generation approaches (Sections 2.4, 6).

A.2 Description & Requirements

This artifact has been executed on a server mounting two NVIDIA L40 GPUs with 48GB VRAM, 1024GB RAM, and two AMD EPYC 7763 CPUs. The server runs Ubuntu 22.04.04 (jammy), with CUDA version 12.4.

A.2.1 Security, privacy, and ethical concerns

Our artifact strictly contains regular Machine Learning preprocessing, training, and testing routines to build NLP models that extract TTPs from CTI reports. They require no destructive steps on the original system, nor disabling of security mechanisms during execution. Therefore, we believe that no specific risks for researchers, practitioners, and evaluators should be encountered.

A.2.2 How to access

Access to our artifact is provided via Zenodo Repository: `https://zenodo.org/records/15608555`. This repository is mirrored by our GitHub repository[1] (code), and our Hugging Face repository[2] (models).

A.2.3 Hardware dependencies

We recommend having at least 48GB of GPU VRAM and 300GB of storage.

A.2.4 Software dependencies

Our artifact partly depends on the DarkBERT[3] model, which requires an access token key by contacting the authors of the paper to be downloaded.

A.2.5 Benchmarks

Our artifacts require the following datasets: TRAM2, and AnnoCTR. Both are provided directly in our repository, in the `dataset` folder.

A.3 Set-up

Our artifact requires Docker[4] installed on the target system. We have tested our artifact with Docker version 25.0.3, build 4debf41. In addition, fine-tuned models and pre-calculated embeddings should be directly downloaded from our Hugging Face repository in Section A.2.2.

[†]Authors contributed equally to this work.

[1]`https://github.com/MarvinBuechel/SoK_CTI_TTP`

[2]`https://doi.org/10.57967/hf/5736`

[3]Jin, Youngjin, et al. "Darkbert: A language model for the dark side of the internet." arXiv preprint arXiv:2305.08596 (2023).

[4]`https://www.docker.com/`

A.3.1 Installation

First, install **Docker** following the instructions reported on the tool website[5].

Second, clone our repository using the link provided in Section A.2.2.

NER. Navigate to the `CTISOK/NER` directory. To install all the required resources, we recommend building the Dockerfile by running: "`$ docker build . -t ner`"

Classification. Make sure to be located in the `CTISOK/classification` folder. Download from our Hugging Face repository the directories `fine_tuned.tgz`, containing our BERT-based fine-tuned models, `local.tgz`, containing some pre-trained models, and `nvidia_embeddings.tgz`, containing pre-calculated embeddings. Decompress them with "`$ tar zxvf archiveName`". The first two folders (`fine_tuned`, `local`) must be located in the root of this sub-project. The content of the second folder must be moved inside the `datasets` directory (after decompressing it you should already see this content inside the folder, otherwise you can run "`$ mv nvidia*.pickle datasets/`"). At last, you should run the following command to build the Docker image: "`$ docker build --build-arg RUN_DEVICE=YOUR_DEVICE -t sok-classification .`", where `YOUR_DEVICE` should point to a GPU (e.g., `cuda`, `cuda:0`, etc.).

Generation. Navigate to the `CTISOK/generation` directory. Build all docker containers and run all experiments by: "`$ docker compose up --build -d`". This automatically downloads the required base models (Meta LLama3.1 8B and GTE-Qwen2-7B for RAG) and involves the installation of an ollama server[6] for the RAG model and a Python environment that starts the Generation experiments automatically.

As runtime inference environment for the generative model, we used the Python library `unsloth`[7] in version 2024.12.4 with the `transformers`[8] library version 4.44.2. Unfortunately, we could no longer reproduce this combination of dependencies, which is why we switched to the newer versions `unsloth==2025.03.19` and `transformers==4.51.1`. This results in the behavior as described in chapter 6.2 - Prompt based, that the LLM suffers from instability and sometimes gets into a loop in which all MITRE IDs are output, which increases the recall, but reduces the precision and accordingly changes the F1 score.

A.3.2 Basic Test

NER. To run the minimal working example, execute the following line: "`$ docker run ner`". This script will test the Docker image by executing the ablation study presented in Table 5 of our work.

Classification. You can run a basic test by executing: "`docker run --gpus all -it sok-classification`". This script checks that models and data are correctly loaded, and perfors a small test by producing part of the results shown in Table 6 of our work.

Generation. To run a minimal test, navigate to the generation folder and execute: "`docker build -t minimal -f Dockerfile.minimal .`" followed by: "`docker run --volume ./experiments:/workspace/finetuning/experiments --gpus all minimal`" This loads a 4-bit quantized model that repeats the "Raw" experiment from Table 9 on the AnnoCTR dataset in "`./experiments/minimal.csv`". Although the quantized model is significantly smaller, it still requires approximately 8GB VRAM.

A.4 Evaluation workflow

Our evaluation workflow is subdivided according to the three sub-projects (NER, Classification, Generation). We also present a subsection for claims that require results from all three approaches (All). For convenience, here, experiments are named after the tables and figures of the paper that show these results.

A.4.1 Major Claims

NER (N).

(N.C1): *Individual pipeline components contribute to improving the recall of NER approaches at the cost of false detections.*

Classification (C).

(C.C1): *Labeled classification approaches achieve higher performances than unlabeled approaches.*

(C.C2): *CTI-specific classification models do not consistently outperform generic models.*

(C.C3): *Data augmentation provides a small margin of improvement for labeled approaches.*

Generation (G).

(G.C1): *Finetuning is the best performing method for generative LLMs and outperforms in-context learning methods.*

(G.C2): *Data augmentation does not replace high-quality datasets.*

(G.C3): *RAG on document-level granularity performs worse than at sentence level.*

All (A).

(A.C1): *Traditional NER approaches outperform both Classification and Generation when no prior information about test data is available and over increasing amount of TTP labels. In general, performances drop with increasing amount of TTP labels.*

[5] https://docs.docker.com/engine/install/

[6] https://github.com/ollama/ollama

[7] https://github.com/unslothai/unsloth

[8] https://github.com/huggingface/transformers

A.4.2 Experiments

(N.E1): [Table 5] [30 human-minutes + 30 compute-minutes]: creates all NER pipelines and runs the ablation study on the TRAM2 and AnnoCTR datasets (presented on Table 5 of our paper).
Execution: Run: "`$ docker run -it ner bash`" in `CTISOK/NER` directory. Run "`$ ./ablation.sh`" inside the container.
Results: Results of all pipelines are stored in the `/NER/experiments/results` directory. They should be compared with results provided in Table 5 of our paper.

(C.E1): [Table 6 and 7] [30 human-minutes + 2 compute-hours]: run the fine-tuned models (labeled approach), and the unlabeled models (with pre-calculated decision thresholds) on the TRAM2 and AnnoCTR datasets. These results are shown in Table 6 and 7 of our paper.
Preparation: Open directory "`CTISOK/classification`". Make sure you have non-empty:
"`fine_tuned/tram_swipe`",
"`fine_tuned/bosch_swipe`", and
"`configs/all_sentence_similarity.json`". Make sure you have "`nvidia-*embeddings.pickle`" inside the "`datasets`" folder.
Execution: First, you need to enter the shell of the docker container. You can do so by running: "`$ docker run --gpus all -it sok-classification bash`". Then, you should run the following commands:
"`$ ./artifact_eval/gen_table_6_annoctr.sh`"
"`$ ./artifact_eval/gen_table_6_tram.sh`"
"`$ ./artifact_eval/gen_table_7.sh`"
"`$ ./artifact_eval/gen_table_7_nvidia.sh`"
Results: Compare Table 6 with the output of the first two scripts. Compare Table 7 with the output of the last two scripts.

(C.E2): See **C.E1**. Such procedure generates the same tables of our original work that support this claim (Tables 6, 7).

(C.E3): [Table 8] [5 human-minutes + 5 compute-minutes]: run the models fine-tuned on augmented datasets and test them on the TRAM2 dataset.
Preparation: Make sure you have the folder named "fine_tuned/data_augmentation" (non-empty).
Execution: Enter the shell of the docker container. Run: "`$ docker run --gpus all -it sok-classification bash`". Then, you should run the following command:
"`$ artifact_eval/gen_table_8.sh`"
Results: Compare the results printed by the script with Table 8 of our paper.

(G.E1): [Table 9] [5 human-minutes + 24 compute-hours]: Runs the model on TRAM2 and AnnoCTR datasets with all the methods presented.
Execution: Run: "`$ docker compose up --build -d`" in the `CTISOK/generation` directory to also execute **G.E2** and **G.E3** experiments.
Results: Results are located at `CTISOK/generation/experiments/table9_methods.csv`.

(G.E2): [Table 10] Trains the base model on the TRAM2 augmented dataset and evaluates it on the TRAM2 test set.
Execution: See **G.E1**.
Results: Results are located at `CTISOK/generation/experiments/table10_augmented_data.csv`.

(G.E3): [Table 11] Runs in-context learning methods on a document-level granularity on TRAM2 and AnnoCTR datasets.
Execution: See **G.E1**.
Results: Results are located at `CTISOK/generation/experiments/table11_document_level.csv`.

(A.E1): [Figure 4] [30 human-minutes + 1 compute-hour]: in this experiment, we produce the data required for obtaining Figure 4 of our paper, which shows the comparison between all approaches on the TRAM2 dataset. Following the color legend proposed in Figure 4, in blue, we explain the parts related to NER, in red, the parts related to classification, in green, the parts related to generation.
Execution (NER): Open directory "NER". Enter the shell of the docker container. Run "`$ docker run -it ner bash`". Then, run the following command: "`$ ./open_set_scenario.sh`"
Results (NER): Results are located in "`results/OpenSet/*.txt`".
Preparation (Classification): Make sure you have "configs/open_set_scenario.json".
Execution (Classification): Open directory "classification". Enter the shell of the docker container. Run: "`$ docker run --gpus all -it sok-classification bash`". Then, run the following command:
"`$ ./artifact_eval/gen_figure_4.sh`"
Results (Classification): The results (F1, Precision, Recall) printed by the script should generate a figure similar to the one presented in Figure 4.
Execution (Generation): Run: "`$ docker compose up --build -d`" in `CTISOK/generation` folder.
Results (Generation): The results are located at `CTISOK/generation/experiments/figure4_open_set.csv`

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: "Whispering Under the Eaves: Protecting User Privacy Against Commercial and LLM-powered Automatic Speech Recognition Systems"

Weifei Jin[†], Yuxin Cao[‡], Junjie Su[†], Derui Wang[§], Yedi Zhang[‡], Minhui Xue[§], Jie Hao[†], Jin Song Dong[‡], Yixian Yang[†]

[†]Beijing University of Posts and Telecommunications [‡] National University of Singapore [§] CSIRO's Data61

A Artifact Appendix

A.1 Abstract

Our paper proposes a framework for protecting the privacy of voice communications, AudioShield, whose core is the Transferable Universal Adversarial Perturbation in the Latent Space (LS-TUAP). This artifact appendix provides a roadmap for reproducing the following **three main claims** in our paper:

- **(C1) Real-time Requirement**: We have implemented the universality of LS-TUAP, which means that our adversarial perturbation is *input-agnostic*, making it effective for any input audio without needing to optimize a separate perturbation for each audio input. This avoids the time-consuming process of optimizing a perturbation for each individual audio input, thus meeting the real-time requirement.
- **(C2) Model-agnostic Requirement**: We have implemented the transferability of LS-TUAP, meaning that our perturbation does not rely on local surrogate models and remains effective for *unseen* ASR models.
- **(C3) High-quality Requirement**: We do not introduce noise in the original audio space, so the adversarial examples generated by our LS-TUAP maintain high audio quality. This is a significant advantage distinguishing our method from existing approaches.

A.2 Description & Requirements

This appendix recommends using the state-of-the-art (SOTA) NN-based ASR model, **Whisper-large-v3** [5], along with 2000 randomly selected audio samples from the full test dataset for reproduction. You can conveniently reproduce our results on your own machine, provided that it meets the following dependencies.

A.2.1 Security, privacy, and ethical concerns

Our artifact does not actively cause any damage to the evaluators' devices.

A.2.2 How to access

- For the latest source code, please download it from our GitHub repository: `https://github.com/WeifeiJin/AudioShield`.
- The complete training dataset (500 samples from LibriSpeech [4]), the test dataset (2000 samples from the VCTK Corpus [6]), and the pretrained **VITS** [2] and **DeepSpeech** [1] models are included in our Zenodo release: `https://doi.org/10.5281/zenodo.14711220`.

Our method requires the **VITS** model and the **DeepSpeech** model during training, both of which are provided in our Zenodo artifact. Additionally, this appendix recommends using the **Whisper-large-v3** model for testing, which can be downloaded using the `download_whisper.py` script or from `https://huggingface.co/openai/whisper-large-v3`.

A.2.3 Hardware dependencies

Our artifact has been tested on the following hardware: an NVIDIA GeForce RTX 4090 GPU and an Intel(R) Xeon(R) Platinum 8352V CPU @ 2.10GHz. We recommend using this hardware configuration for evaluation. If you cannot meet this requirement, please ensure that your GPU has at least 16GB of VRAM.

A.2.4 Software dependencies

The software environment includes Ubuntu 20.04, eSpeak, Python 3.8, CUDA 12.2, and PyTorch 2.2.2. Additionally, all required Python libraries are listed in the `requirements.txt` file of our Github artifact.

A.2.5 Benchmarks

The training dataset (500 samples from LibriSpeech) and the testing dataset (2000 samples from VCTK) are both provided in our artifact.

A.3 Set-up

To prepare the environment for evaluating the artifact, a server with at least the minimum hardware configuration is required. Then, simply follow the commands provided in the GitHub release to set up the necessary environment.

A.3.1 Installation

The installation steps for the required dependencies are detailed in the `README` file of our GitHub-released artifact. Please carefully read its **Setup** section and follow the steps accordingly. You may verify the installation by checking the following key steps:

- Install `eSpeak` using `apt`, a software for text-to-speech.
- Create a Conda environment and install dependencies from `requirements.txt`.
- Build Monotonic Alignment Search for the VITS model.

Additionally, ensure that you have downloaded the required pretrained models: **VITS**, **DeepSpeech**, and **Whisper-large-v3**. Place them in the following directories:

- `pretrained/vits`
- `pretrained/deepspeech`
- `pretrained/whisper-large-v3`

Moreover, place the pickle files (with suffix `.pkl`) of the two datasets in the `datasets` folder. Ensure that these paths are consistent with those specified in `configs/protection.json` (they should match by default, but if you modify them, ensure consistency).

A.3.2 Basic Test

Once the environment is set up, run the `train.py` file from the root directory. If no errors occur, the dependencies have been successfully installed.

A.4 Evaluation workflow

The core parts of the artifact are the following two files, which enable the evaluation of functionality and reproducibility:

- `train.py`: Provides the training process for LS-TUAP, which is the core of our method.
- `eval.py`: Provides the evaluation process for LS-TUAP, and the experimental results will support our main claims.

A.4.1 Major Claims

Please refer to Section A.1 for a detailed recap of our claims. Mainly, we have claimed that:

(C1): Real-time Requirement: *Our LS-TUAP is general and effective for any audio, thus meeting real-time requirements.*

(C2): Model-agnostic Requirement *Our LS-TUAP is not limited to local surrogate models and is effective for unseen models.*

(C3): High-quality Requirement *The adversarial samples we generate maintain high audio quality.*

A.4.2 Experiments

(E1): ***[Training]*** *[10 human-minutes + 30 compute-hour]: This stage prepares the LS-TUAPfor the subsequent evaluation phase.*
How to: *You can directly run `python train.py` to train the LS-TUAP using default parameters.*
Preparation: *Make sure you have the training dataset prepared, and that the pretrained VITS and DeepSpeech models are downloaded and placed in the corresponding folders as described in Section A.3.1.*
Execution: *We recommend training for 3 epochs, although this process will take approximately 30 hours. For the convenience of quick evaluation, we have provided a pretrained LS-TUAP model in our GitHub artifact, and you can opt to use the pretrained LS-TUAP for evaluation directly.*
Results: *You will get a well-trained perturbation file named `LS-TUAP.pth`.*

(E2): ***[Evaluation]*** *[1 human-minute + 30 compute-minutes]: This is the critical evaluation process.*
How to: *You can directly run `python eval.py` to evaluate AudioShield using the default parameters.*
Preparation: *Ensure that the test dataset is ready, and that the test model `Whisper-large-v3` is downloaded and placed in the appropriate folder as described in Section A.3.1.*
Execution: *We recommend training for 3 epochs, although this process will take approximately 30 hours. For the convenience of quick evaluation, we have provided a pretrained LS-TUAP model in our GitHub artifact, and you can opt to use the pretrained LS-TUAP for evaluation directly.*
Results: *Combine the above evaluation process and results, and check whether they support the three main claims in Section 1 as described below:*

- **(C1) Real-time Requirement**: *This means universality—our LS-TUAP is effective for any input audio. You can observe that, during the evaluation process, the same LS-TUAP is used for all input audio, eliminating the need to optimize a perturbation*

for each audio individually, thus meeting real-time requirements.

- **(C2) Model-agnostic Requirement**: *We trained the ASR model with DeepSpeech, and tested it with Whisper. The PSR, CER, and WER results you obtain in the evaluation should generally match those in **Table 6** (due to the inherent randomness of the Whisper model, there may be some variability in results each time it is run).*
- **(C3) High-quality Requirement**: *We used the SOTA speech quality assessment model NISQA [3] for evaluation. The results should generally match those presented in **Table 7**, with our NISQA score always above 2.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182. PMLR, 2016.

[2] Jaehyeon Kim, Jungil Kong, and Juhee Son. Conditional variational autoencoder with adversarial learning for end-to-end text-to-speech. In *International Conference on Machine Learning*, pages 5530–5540. PMLR, 2021.

[3] Gabriel Mittag, Babak Naderi, Assmaa Chehadi, and Sebastian Möller. Nisqa: A deep cnn-self-attention model for multidimensional speech quality prediction with crowdsourced datasets. *arXiv preprint arXiv:2104.09494*, 2021.

[4] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

[5] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International conference on machine learning*, pages 28492–28518. PMLR, 2023.

[6] Junichi Yamagishi, Christophe Veaux, Kirsten MacDonald, et al. Cstr vctk corpus: English multi-speaker corpus for cstr voice cloning toolkit (version 0.92). *University of Edinburgh. The Centre for Speech Technology Research (CSTR)*, 2019.





USENIX Security '25 Artifact Appendix: When Translators Refuse to Translate: A Novel Attack to Speech Translation Systems

Haolin Wu[1,2], Chang Liu[3], Jing Chen[1,2], Ruiying Du[1,2], Kun He[1,2], Yu Zhang[1,2], Cong Wu[4], Tianwei Zhang[4], Qing Guo[5], Jie Zhang[5]

[1]School of Cyber Science and Engineering, Wuhan University, China

[2]Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, China

[3]School of Cyber Science and Technology, University of Science and Technology of China, China

[4]College of Computing and Data Science, Nanyang Technological University, Singapore

[5]CFAR and IHPC, Agency for Science, Technology and Research, Singapore

A Artifact Appendix

A.1 Abstract

This artifact appendix accompanies our paper and provides all necessary resources to evaluate the proposed attack. It includes detailed instructions to reproduce the major claims, such as the motivation for our work, as well as the implementation of sample-level and universal attacks. Additionally, we outline the hardware, software, and configurations required for successful evaluation.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact poses no security or privacy risks to evaluators' systems. However, we advise evaluators to handle the generated adversarial examples responsibly.

A.2.2 How to access

We host our artifacts at Zenodo: https://doi.org/10.5281/zenodo.14735820.

A.2.3 Hardware dependencies

Running our code requires an NVIDIA GPU with at least 24GB of VRAM. For reference, we used an NVIDIA GeForce RTX 4090 GPU in our experiments. No specific hardware beyond the GPU is necessary; our setup included Intel(R) Xeon(R) Gold 6133 CPU and 256GB of RAM.

A.2.4 Software dependencies

We conducted our experiments on Ubuntu 20.04.3 LTS with NVIDIA Driver Version 525.105.17, CUDA Version 12.0 and Python 3.10.13. Details on additional software dependencies and information about datasets and models used in our experiments can be found in the README file.

A.2.5 Benchmarks

Datasets. We use several popular speech datasets: Common Voice, TIMIT, and LibriSpeech. Additionally, we include two widely used speech translation datasets: MuST-C and Europarl-ST. The download URLs and dataset splits are detailed in the README file provided with our artifacts.

Models. We target the Seamless family of models, specifically evaluating Seamless M4T v2 Large and Seamless Expressive. Model details and download instructions are available in their official repository (https://github.com/facebookresearch/seamless_communication). Additionally, we use the 51-languages-classifier model from Hugging Face for automated language classification. Commands for downloading it are included in the README file.

A.3 Set-up

A.3.1 Installation

Here are the steps to install the required dependencies and run our code. For detailed instructions and commands, please check the README file provided with our artifacts.

1. Install `seamless_communication` and other dependencies by running commands in the README file.

2. Prepare the pretrained Seamless models by following the README instructions. Note that the Hugging Face library will automatically download the Seamless M4T v2 Large model, but you must complete the Meta request form to access the Seamless Expressive model.

3. Download the datasets with the provided URLs in the README file. Pay attention to the path where you save the datasets and update it in the config file if necessary.

A.3.2 Basic Test

We prepared a python script to run a simple functionality test. The script can be run using the following command:

```
python BasicTest.py --gpu_id 0
```

This script is designed to generate an adversarial example with sample-level attack. Upon successful execution, the script is expected to present a text output beginning with "The basic test has been successfully completed."

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): State-of-the-art multilingual speech translation systems exhibit an inherent vulnerability, where they tend to output content in the source language instead of the target language. This claim is demonstrated through experiment (E1) in Section 3, "Motivation," of our paper, with results illustrated in Figure 6 and Figure 17.

(C2): Sample-level untranslation attacks effectively generate adversarial examples that deceive models into outputting the original speech content in the source language with high success rates. This is supported by experiment (E2) in Section 6.2, "Sample Level Attack," of our paper, with results reported in Table 2.

(C3): Universal untranslation attacks can be used to train a universal adversarial perturbation that manipulates the model to output the original speech content in the source language when appended to the original speech. This is proven by experiment (E3) in Section 6.3, "Universal Attack," of our paper, with results shown in Table 6.

A.4.2 Experiments

(E1): [Motivation] [25 human-minutes + 5 compute-minutes + 24GB RAM]: In this experiment, we demonstrate the vulnerability of state-of-the-art multilingual speech translation systems to untranslation attacks. We show that these systems tend to output the original speech content in the source language. The details of this experiment can be found in the "Vulnerability of Untranslation" section of our paper. The expected outcome is to observe that the average logits value of source language tokens is significantly higher than the average logits value of target language tokens.
Preparation: Install the required dependencies and download the datasets and models as described in Section A.3.
Execution: Run the following command:
`python MotivationExp.py --gpu_id 0`
Results: The expected outcome is to observe that the average logits value for tokens of the original content in the source language is significantly higher than for other tokens. Additionally, a plot of the distribution of logits values similar to Figure 6 and Figure 17 in our paper will be generated in the `./plot` folder.

(E2): [Sample-level Attack] [25 human-minutes + 10 compute -hours + 24GB RAM]: In this experiment, we use the proposed sample-level attack to generate adversarial examples that manipulate the models into outputting the original speech content in the source language. Details can be found in Section 6.2, "Sample Level Attack," of our paper. The expected outcome is to observe a high success rate of the attack.
Execution: Run the following command:
`python SampleLevelAttackEval.py --gpu_id 0`
Results: The expected outcome is to observe the attack process for each sample, with results including success rate, time cost, and other details.

(E3): [Universal Attack] [25 human-minutes + 36 compute-hours + 24GB RAM]: In this experiment, we use the proposed universal attack to train a universal adversarial perturbation that manipulates the models into outputting the original speech content in the source language by appending the perturbation to the original speech. We then evaluate the success rate of the perturbation on an evaluation dataset. Details can be found in Section 6.3, "Universal Attack," of our paper. The expected outcome is to observe a high success rate of the attack.
Execution: Run the following command to train the universal perturbation:
`python UniversalAttackTrain.py --gpu_id 0`
Then, run the following command to evaluate the universal perturbation:
`python UniversalAttackEval.py --gpu_id 0`
Results: During the training process, the expected outcome is to observe the progress of the universal perturbation training. After training, the perturbation will be saved in the `./universal_perturbation` folder. During the evaluation process, the expected outcome is to observe the attack process of the universal perturbation.

Please note that you can always use the -h flag with each script to check the available options and parameters. Also, refer to the README file in the artifacts for detailed instructions and commands.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: VAPD: An Anomaly Detection Model for PDF Malware Forensics with Adversarial Robustness

Side Liu
Wuhan University

Jiang Ming
Tulane University

Yilin Zhou
Wuhan University

Jianming Fu
Wuhan University

Guojun Peng
Wuhan University

A Artifact Appendix

A.1 Abstract

In this work, we propose VAPD, a reconstruction-based anomaly detection model designed with dual forensic objectives: (1) identifying PDF malware through reconstruction errors between the input and output, and (2) localizing anomalous regions within the document. We evaluate VAPD on multiple datasets, samples, achieving an accuracy of 99.54%, outperforming existing anomaly detection methods. Furthermore, we assess the robustness of VAPD using four adversarial attack frameworks in both feature space and problem space, demonstrating its strong resilience against adversarial manipulations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact does not involve steps such as vulnerability reproduction and malicious code execution. All data has been processed, and there will be no risks to computer security and privacy.

A.2.2 How to access

The evaluation preprocessed datasets and code used in this artifact are available at https://zenodo.org/records/15603773 and https://zenodo.org/records/16732430. This artifact can be accessed publicly.

A.2.3 Hardware dependencies

1. At least 250 GB of available disk space;
2. At least 32 GB of RAM;
3. A GPU of 1080 Ti or higher is recommended, with more than 11 GB of VRAM (Optional).

A.2.4 Software dependencies

1. Python 3.10 or higher;
2. CUDA 11.8 or higher (Optional);
3. Other Python package dependencies and versions can be found in requirements.txt.

A.2.5 Benchmarks

1. Contagio Baseline Dataset;
2. Baseline models such as RTM, KNN, and DeepSVDD have been placed in the artifact.

A.3 Set-up

A.3.1 Installation

Running the artifact requires installing Python, along with the corresponding Python packages and a compatible CUDA version (CUDA is optional — required for training, but GPU and CUDA may not be necessary for inference/testing).

Python First, please download Python 3.10 or a later version from the official Python website: Python 3.10.17.

Python Package Dependencies) We recommend using Virtualenv to create a virtual environment. First, install Virtualenv by running:

```
> pip3 install virtualenv
```

Then, create a virtual environment:

```
> virtualenv venv
```

Activate the virtual environment:

```
> source venv/bin/activate
```

Finally, install the required packages provided in https://zenodo.org/records/16732430:

```
> pip3 install –r requirements.txt
```

CUDA (Optional) If a GPU is available, you may optionally install the CUDA Toolkit from https://developer.nvidia.com/cuda-toolkit to enable GPU acceleration during training, which can significantly speed up the process.

If no GPU is available, the artifact can still be run using the CPU for testing or inference purposes.

A.3.2 Basic Test

Navigate to the *vapd_src* directory first, and execute `test.py` like the following:

```
> python test.py
```

And, it will ouput:

```
usage: test.py [-h] -dataset DATASET -label
LABEL -model MODEL -model_path MODEL_PATH
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): VAPD demonstrates outstanding classification performance across multiple PDF datasets (E1).

(C2): VAPD exhibits strong adversarial robustness against various adversarial attack settings (E2).

(C3): VAPD is capable of localizing anomalous objects within PDF files. (E3).

A.4.2 Experiments

The experiments are divided into three parts:

(E1): Detection on D1, D2, and D3.

(E2): Adversarial experiments against adversarial attacks.

(E3): Localization experiments within PDF.

E1. Detection on baseline and extended dataset [5 human-minutes + 5 compute-minutes]

Navigate to the *vapd_src* directory:

```
> cd vapd_src
```

The evaluation script is test.py, and the usage of test.py is as follows:

```
usage: test.py [-h] -dataset DATASET -label
LABEL -model MODEL -model_path MODEL_PATH
```

- -dataset: the path to the preprocessed dataset file
- -label: lable file
- -model: the model to be evaluated, VAPD | KNN | AE | DeepSVDD | RTM

Please run the following command to evaluate VAPD on the D1 dataset and it will output the result on console:

```
> python test.py --dataset ./data/D1_test.npy --label
./data/D1_label.npy --model VAPD --model_path
./models/vapd_weights.pkl
```

As for D2, please run the following command:

```
> python test.py --dataset ./data/D2_test.npy --label
./data/D2_label.npy --model VAPD --model_path
./models/vapd_weights.pkl
```

As for D3, please run the following command:

```
> python test.py --dataset ./data/D3_test.npy --label
./data/D3_label.npy --model VAPD --model_path
./models/vapd_weights.pkl
```

E2. Adversarial experiments against adversarial attacks. [2 human-minutes + 60 compute-minutes]

For the gradient descent attack, first navigate to the adversarial attack directory:

```
> cd adversarial\ attack/
```

Then simply run GradAttack.py as following:

```
> python GradAttack.py
```

This script will print the process in the console and generate a CSV file to save the final results.

For MalGAN, first navigate to the malgan directory

```
> cd adversarial\ attack/malgan
```

Then run the following command. The console will display the progress, and a figure will be generated at the end as the result.

```
> python cus_malgan.py
```

Since the evaluation of EvadeML and reverse mimicry involves setting up a distributed Cuckoo sandbox cluster, we are currently unable to provide ready-to-run containers or virtual machines. Moreover, the dynamic verification process is extremely time-consuming, with a full run potentially taking over four weeks. Therefore, we have only implemented adversarial evaluations in the feature space, which are much easier to run and reproduce.

E3. Localization experiments within PDF [5 human-minutes + 5 compute-minutes]

We have prepared a subset of approximately 100 samples from the test set for evaluating localization. To use it, run the following command. The value after –index should be an integer between 0 and 100. After execution, the top-5 localization results will be printed to the console.

```
> python locate.py --index 12
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: High Stakes, Low Certainty: Evaluating the Efficacy of High-Level Indicators of Compromise in Ransomware Attribution

Max van der Horst
Delft University of Technology

Ricky Kho
Sogeti

Olga Gadyatskaya
Leiden University

Michel Mollema
Northwave Cybersecurity

Michel van Eeten
Delft University of Technology

Yury Zhauniarovich
Delft University of Technology

A Artifact Appendix

A.1 Abstract

The artifacts included for this paper are located in a Zenodo repository. The Zenodo repository contains the individual appendices of the paper, such as the research protocol, the codebook, the observed techniques per investigated threat actor, and the CISA reports. Additionally, the gathered data on the individual threat actors as well as the code to analyze this data, is included. To reproduce the results of the paper, the code in `code.zip` can be run by following the steps in `README.md`.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The code included in the artifacts can exclusively process the included data and will not pose any security concerns for the users. All data has been anonymised where applicable, and all artifacts have been produced following the ethical considerations described in the paper.

A.2.2 How to access

All artifacts have been included in the Zenodo repository on https://zenodo.org/records/14732551.

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

The artifacts can be used on any operating system. However, Python 3.11+ and Python Poetry[1] is used. The file `pyproject.toml` includes all individual dependencies that are installed by Poetry. To install these, follow the steps included in the `README.md` file. Moreover, either the VSCode IDE or Jupyter Notebook should be installed, though the notebook is best evaluated using VSCode.

[1] https://python-poetry.org

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Before the file `data_analysis.ipynb` can be run, the system must have Python 3.11+ available and be able to run Poetry – a dependency manager for Python projects. Poetry can be installed through pipx: `pipx install poetry`. After installation of Poetry, the dependencies can be installed with the command `poetry install -no-root`. The notebook can best be evaluated in VSCode. If the reviewer chooses to use Jupyter Notebook (which is another option), appropriate dependencies have to be installed to support it.

A.3.2 Basic Test

After opening the notebook, the first cell in the notebook for the definition of the constants can be run to verify that the system is ready to execute the other cells.

A.4 Evaluation Workflow

The artifacts can be evaluated using the following steps.

1. Download the artifacts `code.zip` and `data.zip` from the Zenodo repository.
2. Ensure that the VSCode IDE is installed.
3. Ensure that all dependencies have been installed using Python Poetry.

4. Open `data_analysis.ipynb` in VSCode to open the data analysis notebook.
5. Ensure the notebook has access to the contents of `data.zip` in `../data` (relative to the working location of the notebook).
6. Execute all cells in the data analysis notebook.
7. The analysis results will be stored in `../results`, with the LaTeXtables being located in the `tabs/` subdirectory.

A.4.1 Major Claims

The major claims in this section are mainly discussed in Sections 4.2 and 5.2 of the paper. As explained in the Ethics section at the end of the paper, we only include data on the quantitative analysis for review as we cannot share some data artifacts (transcripts and reports).

High Overlap Across Different RTAs. The paper mentions a high overlap of TTPs across different RTAs. This is proven by the overlap similarity matrix in Table 6 of the paper. This table can be generated by following the "Analysis" section of the notebook, which is computed by calculating the overlap similarity for all data points.

Inconsistent TTP Usage Within the Same RTA. Attacks attributed to the same actor only shared, on average, about 37% of their TTPs. This is proven by following the steps in the "Similarity between Variants of the same TA" section in the notebook to generate the content of Table 5 in the paper.

Negative Silhouette Scores. By executing the first cell of the "Company Data" section in the notebook, the silhouette score calculated based on the clusters of RTAs is negative. This indicates that TTP sets for different ransomware actors do not define clearly separable clusters.

Coverage Gaps and Fragmentation. By executing the code belonging to the company data and the CISA data in the notebook, it can be seen that there are significantly differing overlap scores for the same RTA. Given that this is a consistent result for various actors and not an anomaly, it can be suggested that not one party has a complete overview of the data belonging to RTAs.

A.4.2 Experiments

Please find the steps to reproduce the results for the major claims below. Each section of the results is preceded by the first five steps as described in the Evaluation Workflow.

1. Download the artifacts `code.zip` and `data.zip` from the Zenodo repository.
2. Ensure that the VSCode IDE is installed.
3. Ensure that all dependencies have been installed using Python Poetry.
4. Open `data_analysis.ipynb` in VSCode to open the data analysis notebook.
5. Ensure the notebook has access to the contents of `data.zip` in `../data` (relative to the working location of the notebook).

High Overlap Across Different RTAs [Human time: 5 minutes, compute time: < 1 second]

1. Ensure the execution of the cells in the "Analysis" section of the notebook to define the functions.
2. Execute the cell "Similarity between Different Families in Company Reports".
3. The notebook will output the mean overlap similarity for different RTAs in the company reports, and a TeX table will be generated in the file `overlap_sim_of_tas_comp.tex`.
4. Overlap similarity is expressed as a value between 0 and 1, with 0 meaning that there is no overlap and 1 indicating identical values.

Inconsistent TTP Usage Within the Same RTA [Human time: 5 minutes, compute time: < 1 second]

1. Ensure the execution of the cells in the "Analysis" section of the notebook to define the functions.
2. Execute the cell "Similarity between Variants of the Same TA".
3. The notebook will output the results in the file `tavar_sim_table_comp.tex`.
4. The overlap similarity in the table is expressed as a value between 0 and 1, with 0 meaning that there is no overlap and 1 indicating identical values.

Negative Silhouette Scores [Human time: 5 minutes, compute time: < 1 second]

1. Ensure the execution of the cells in the "Analysis" section of the notebook to define the functions.
2. Execute the first cell of the section "Company Data".
3. The notebook will output the silhouette scores using both Euclidean and cosine distance metrics.
4. Silhouette score is a value between -1 and 1. A lower score indicates poorly defined clusters and a higher score indicates well-defined clusters. The silhouette score is negative for our data, indicating poorly defined clusters.

Coverage Gaps and Fragmentation [Human time: 5 minutes, compute time: < 1 second]

1. Ensure the execution of all cells in the notebook up until the "Comparison of Techniques in CISA and Company reports" cell.
2. The notebook will output the mean overlap similarity and generate a TeX table in the file `comp_cisa_sim_analysis.tex`.
3. Comparing the table generated in this step to the `overlap_sim_of_tas_comp.tex` table from the first major claim, it can be seen that the overlap similarities per TA are very different. This indicates a coverage gap and fragmentation.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: A Tale of Two Worlds, a Formal Story of WireGuard Hybridization

Pascal Lafourcade[1], Dhekra Mahmoud[1], Sylvain Ruhault[2] and Abdul Rahman Taleb[2]

[1]Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Mines Saint-Etienne, LIMOS, 63000 Clermont-Ferrand, France
[2]Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI), France

A Artifact Appendix

A.1 Abstract

These artifacts accompany research paper "A Tale of Two Worlds, a Formal Story of WireGuard Hybridization". They allow to reproduce all results described in this paper, concerning symbolic evaluation of security properties and implementation of protocols related to hybridization of WireGuard. These artifacts are composed of two folders. The first folder **artifacts_implementation** concerns our Rust implementation of WireGuard, PQ-WireGuard⋆ and Hybrid-WireGuard. The second folder **artifacts_evaluation** concerns our symbolic analysis of WireGuard (with fix for anonymity based on psk), PQ-WireGuard, PQ-WireGuard⋆ and Hybrid-WireGuard, with the help of TAMARIN, PROVERIF and DEEPSEC verifiers.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Our artifacts are available on a permanent public repository[1]. To access them, execute (MacOS may require to remove \):

```
record_url=$(curl -Ls -o /dev/null -w '%{url_effective}' \
https://doi.org/10.5281/zenodo.15551056)
curl -L -o artifacts-usenix-2025-wireguard-hybridization.zip \
"$record_url/files/artifacts-usenix-2025-wireguard-hybridization."\
"zip?download=1"
unzip artifacts-usenix-2025-wireguard-hybridization
cd artifacts-usenix-2025-wireguard-hybridization
```

A.2.3 Hardware dependencies

Our artifacts do not require any specific hardware. All results are reproducible on a standard laptop with 12 cores of CPU 1.8 GHz and 16 Go of RAM.

[1] https://doi.org/10.5281/zenodo.15551056

A.2.4 Software dependencies

Our artifacts require the use of Docker Engine[2]. For our tests, we used Docker Engine version 28.2.2 and we successfully tested Docker deployment on both Linux Ubuntu 24.04.2 LTS and MacOS Sequoia 15.4. We use TAMARIN release version 1.10.0, Maude version 3.5, PROVERIF version 2.05, DEEPSEC version 2.0.2, Python version 3.12.3, Package sympy, GNU parallel, Clang version 18.1.3 and Rust version 1.87.0. All these dependencies are installed automatically when building Dockerfile (subsubsection A.3.1).

A.2.5 Benchmarks

Our artifacts allow to reproduce similar benchmarks as the one given in our paper (Table 9), to compare the efficiency of WireGuard, PQ-WireGuard⋆ and Hybrid-WireGuard. Our benchmarks focus on the construction of InitHello message by Initiator, and on the processing of InitHello and construction of RespHello by Responder. Execution time is averaged on a chosen number of executions of these operations, that is a parameter of the benchmark (subsubsection A.4.2).

A.3 Set-up

A.3.1 Installation

Before building Docker, it is adviced to perform a complete cleanup of Docker environment. For this, execute (warning: this cleanup stops and deletes *all* Docker containers, images, volumes, networks):

```
sh run_docker_clean.sh
```

Then, execute:

```
docker buildx build --platform linux/x86_64 -t hyb-wg .
```

This should build the Docker image with all dependencies (subsubsection A.2.4). Docker build takes around 30 minutes.

[2] https://docs.docker.com/engine/install

A.3.2 Basic Test

To check Docker image, execute:

```
docker run -it hyb-wg
```

Then inside Docker, execute:

```
sh run_basic_test.sh
```

This should launch, successively, TAMARIN, PROVERIF, DEEPSEC verifiers and Cargo Rust, hence their respective versions shall successively appear (subsubsection A.2.4). Furthermore, this basic test launches evaluations of trace properties (resistance against unilateral and bilateral unknown key share attacks, session uniqueness, agreement and secrecy, including perfect forward secrecy) for PQ-WireGuard* protocol. A set of CNFs shall be printed, as in Table 2, equal to the ones described in Table 4 of our paper, for PQ-WireGuard* protocol. This basic test shall finish in less than 2 minutes.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We propose a Rust implementation of protocols WireGuard, PQ-WireGuard*, Hybrid-WireGuard. This is proven by the experiment (**E1**) described in Section 7.2 of our paper, with results reported in Table 9 of our paper.

(C2): We propose, with TAMARIN, PROVERIF and DEEPSEC verifiers, symbolic proofs of trace properties (session uniqueness, resistance against UKS attacks, message agreement and secrecy, including PFS) and observational equivalence properties (anonymity, strong secrecy), for WireGuard (with fix for anonymity based on psk), PQ-WireGuard, PQ-WireGuard* and Hybrid-WireGuard. This is proven by experiments (**E2**), (**E3**), (**E4**), described in Sections 4, 5, 6 of our paper, whose results are reported in Table 4 of our paper.

A.4.2 Experiments

(E1): *[Efficiency evaluation] [1 human-minutes + 10 compute-minutes]: based on Rust implementation, build an application that mesures efficiency of construction of InitHello message by Initiator, and processing of InitHello and construction of RespHello message by Responder.*
Preparation: Inside Docker image (subsubsection A.3.2), in folder `artifacts_implementation`, file `run_all.sh` contains line `cargo run - -b 100`. Value `100` is the chosen number for the benchmarks, as described in subsubsection A.2.5.
Execution: Run `sh run_all.sh`.
Results: Printed results shall be as in Table 1, where number of executions is the one chosen for benchmarks (subsubsection A.2.5), message sizes correspond to Table 8 of our paper and timings correspond to Table 9 of our paper.

Table 1: Results for Experiment (**E1**).

```
Average time over 100 executions

WireGuard:
InitHello message size: 196 bytes
RespHello message size: 140 bytes
InitHello construction time: 0.230 ms (std = 0.061 ms)
InitHello consumption time + RespHello construction time: 0.507 ms (
      std = 0.148 ms)

PQ-WireGuard*: (static kem: Classic-McEliece-460896, ephemeral kem:
      ML-KEM-512)
InitHello message size: 1124 bytes
RespHello message size: 1032 bytes
InitHello construction time: 0.326 ms (std = 0.168 ms)
InitHello consumption time + RespHello construction time: 0.612 ms (
      std = 0.148 ms)

Hybrid-WireGuard: (static kem: Classic-McEliece-460896, ephemeral kem
      : ML-KEM-512)
InitHello message size: 1156 bytes
RespHello message size: 1064 bytes
InitHello construction time: 0.481 ms (std = 0.074 ms)
InitHello consumption time + RespHello construction time: 1.211 ms (
      std = 0.178 ms)
```

(E2): *[Trace properties evaluation] [1 human-minutes + 4 compute-hours]: based on* SAPIC$^+$ *files, generate all* PROVERIF *and* TAMARIN *files and evaluate trace properties (session uniqueness, resistance against UKS attacks, message agreement and secrecy, including PFS) for WireGuard (with fix for anonymity based on* psk*), PQ-WireGuard, PQ-WireGuard*, Hybrid-WireGuard.*
Preparation: Inside Docker image (subsubsection A.3.2), in folder `artifacts_evaluation`, file `run_trace_properties.sh` contains all necessary commands that allows to, sequentially, perform all evaluations. These use PROVERIF verifier, for all protocols, and TAMARIN verifier for WireGuard (with fix for anonymity based on psk) and Hybrid-WireGuard. We did not use TAMARIN verifier for PQ-WireGuard and PQ-WireGuard* as these protocols do not rely on `diffie-hellman` builtin.
Execution: Run `sh run_trace_properties.sh`.
Results: Printed results shall be as in Table 2. These results correspond to Table 4 of our paper. To check TAMARIN results for WireGuard (with fix for anonymity based on psk), execute:

```
cd wireguard_with_fix_psk/trace_properties/tamarin
cat *.spthy.log | grep "verified\|falsified"
```

All evaluated properties shall be `verified`. To check TAMARIN results for Hybrid-WireGuard, execute:

```
cd hybrid_wireguard/trace_properties_necessary_conditions
cat *.spthy.log | grep "verified\|falsified"
```

All evaluated properties shall be `verified`. Finally, check sufficient conditions for Hybrid:

```
cd hybrid_wireguard/trace_properties_sufficient_conditions
sh run_check-pv.sh
```

All queries shall be `false` except for `uuks` and `buks` which shall be `true`.

WireGuard (with fix for anonymity based on psk)

```
bilateral: ∅
unilateral_initiator: ∅
unilateral_responder: ∅
uniqueness_initiator: ∅
uniqueness_responder: ∅
agreement_inithello: psk & (dhsisr | sic | src)
agreement_rechello: psk & (src | eic) & (dhsisr | sic | src)
agreement_confirm: psk & (sic | erc) & (dhsisr | sic | src)
secrecy_isk7: psk & (src | eic) & (dhsisr | sic | src)
secrecy_rsk7: psk & (sic | erc) & (dhsisr | sic | src)
secrecy_mul7: psk & (sic | erc) & (src | eic) & (eic | erc) & (dhsisr
       | sic | src)
secrecy_isk7pfs: psk & (sic | erc) & (src | eic) & (eic | erc) & (
      dhsisr | sic | src)
secrecy_rsk7pfs: psk & (sic | erc) & (src | eic) & (eic | erc) & (
      dhsisr | sic | src)
secrecy_mul7pfs: psk & (sic | erc) & (src | eic) & (eic | erc) & (
      dhsisr | sic | src)
```

PQ-WireGuard

```
uniqueness_initiator: ∅
bilateral: eipq | re
uniqueness_responder: ∅
unilateral_initiator: psk & (sipq | rr) & (sipq | sigr) & (eipq | re)
secrecy_isk7: psk & (srpq | ri) & (srpq | sigi)
unilateral_responder: psk & (srpq | ri) & (srpq | sigi) & (eipq | re)
agreement_inithello: psk
secrecy_rsk7: psk & (sipq | rr) & (sipq | sigr)
agreement_rechello: psk & (srpq | ri) & (srpq | sigi)
agreement_confirm: psk & (sipq | rr) & (sipq | sigr)
secrecy_isk7pfs: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (
      srpq | sigi) & (eipq | re)
secrecy_mut7: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (srpq
       | sigi) & (eipq | re)
secrecy_rsk7pfs: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (
      srpq | sigi) & (eipq | re)
secrecy_mut7pfs: psk & (sipq | rr) & (sipq | sigr) & (srpq | ri) & (
      srpq | sigi) & (eipq | re)
```

PQ-WireGuard⋆

```
unilateral_responder: ∅
bilateral: ∅
uniqueness_responder: ∅
uniqueness_initiator: ∅
unilateral_initiator: ∅
secrecy_isk7: psk & (srpq | ri)
agreement_confirm: psk & (sipq | rr)
agreement_rechello: psk & (srpq | ri)
secrecy_rsk7: psk & (sipq | rr)
secrecy_mut7: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
agreement_inithello: psk
secrecy_isk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
secrecy_mut7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
secrecy_rsk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re)
```

Hybrid-WireGuard

```
unilateral_responder: ∅
unilateral_initiator: ∅
bilateral: ∅
uniqueness_initiator: ∅
uniqueness_responder: ∅
secrecy_isk7: psk & (srpq | ri) & (src | eic) & (dhsisr | sic | src)
agreement_inithello: psk & (dhsisr | sic | src)
secrecy_rsk7: psk & (sipq | rr) & (sic | erc) & (dhsisr | sic | src)
agreement_rechello: psk & (srpq | ri) & (src | eic) & (dhsisr | sic |
       src)
agreement_confirm: psk & (sipq | rr) & (sic | erc) & (dhsisr | sic |
      src)
secrecy_mut7: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic |
      erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
secrecy_isk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic
       | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
secrecy_rsk7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic
       | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
secrecy_mut7pfs: psk & (sipq | rr) & (srpq | ri) & (eipq | re) & (sic
       | erc) & (src | eic) & (eic | erc) & (dhsisr | sic | src)
```

Table 2: Results for Experiment (**E2**).

(E3): *[Anonymity evaluation] [1 human-minutes + 4 compute-hours]: evaluate all* ProVerif *and* DeepSec *files that model anonymity for WireGuard (with fix for anonymity based on* psk*), PQ-WireGuard, PQ-WireGuard⋆, Hybrid-WireGuard.*

Preparation: Inside Docker image (subsubsection A.3.2), in folder `artifacts_evaluation`, file `run_anonymity.sh` contains all necessary commands that allows to, sequentially, evaluate all files for anonymity for all protocols. These evaluations use ProVerif verifier, for all protocols, and DeepSec verifier for PQ-WireGuard and PQ-WireGuard⋆. We could not use DeepSec verifier for WireGuard (with fix for anonymity based on psk) and Hybrid-WireGuard as DeepSec does not handle exponentiation.

Execution: Run `sh run_anonymity.sh`.

Results for WireGuard (with fix for anonymity based on psk). For ProVerif, execute:

```
cd wireguard_with_fix_psk/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `initiator`, `-Eic`, `-Psk`, `-Src` and for files `responder`, `-Eic`, `-Psk`, `-Src` and shall be `Observational equivalence is true` for files `initiator`, `-no-reveal`, `-Erc-Sic` and for files `responder`, `-no-reveal`, `-Erc-Sic`.

Results for PQ-WireGuard. For DeepSec, execute:

```
cd pq_wireguard/equivalence_properties/Anonymity
sh run_check-dps.sh
```

Results shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for all files.

Results for PQ-WireGuard⋆. For DeepSec, execute:

```
cd pq_wireguard_star/equivalence_properties/Anonymity
sh run_check-dps.sh
```

Results shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard_star/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `initiator`, `-Psk`, `-Ri`, `-Rr`, `-Siq`, `-Srq` and for files `responder`, `-Psk`, `-Ri`, `-Srq` and shall be `Observational equivalence is true` for file `initiator`, `-Eiq-Re` and for file `responder`, `-Siq-Rr-Eiq-Re`.

Results for Hybrid-WireGuard. For ProVerif, execute:

```
cd hybrid_wireguard/equivalence_properties/Anonymity
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `initiator, -Psk, -Eic-Ri, -Eic-Srq, -Erc-Siq, -Erc-Rr, -Sic-Rr, -Sic-Siq, -Src-Ri, -Src-Srq` and for files `responder, -Psk, -Eic-Ri, -Src-Ri, -Src-Srq, -Src-Eic` and shall be `Observational equivalence is true` for files `initiator, -all-dh, -all-pq, -no-reveal, -Sic -Erc-Srq-Eiq-Ri-Re, -Sic-Src-Eic-Erc-Eiq-Re, -Src-Eic-Siq-Eiq-Rr-Re` and for files `responder, -all-dh, -all-pq, -no-reveal, -Sic -Erc-Siq-Srq-Eiq-Ri-Rr-Re, -Sic-Src-Eic-Erc -Siq-Eiq-Rr-Re`.

(E4): *[Strong secrecy evaluation] [2 human-minutes + 10 compute-hours]: evaluate all* ProVerif *and* DeepSec *files that model strong secrecy for WireGuard (with fix for anonymity based on* psk*), PQ-WireGuard, PQ-WireGuard⋆, Hybrid-WireGuard.*

Preparation: Inside Docker image (subsubsection A.3.2), in folder `artifacts_evaluation`, file `run_strong_secrecy.sh` contains all necessary commands that allows to, sequentially, perform all evaluations. These use ProVerif verifier, for all protocols, and DeepSec verifier for PQ-WireGuard and PQ-WireGuard⋆. We could not use DeepSec verifier for WireGuard (with fix for anonymity based on psk) and Hybrid-WireGuard as DeepSec does not handle exponentiation.

Execution: Run `sh run_strong_secrecy.sh`.

Results for WireGuard (with fix for anonymity based on psk). For ProVerif, execute:

```
cd wireguard_with_fix_psk/equivalence_properties/Strong-
     Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-Src, -Psk-DH-Eic, -Psk-Sic-Eic` and shall be `Observational equivalence is true` for files `-Psk-Eic-Erc, -Psk-Sic-Erc, -Sic-Src-Eic-Erc`.

Results for PQ-WireGuard. For DeepSec, execute:

```
cd pq_wireguard/equivalence_properties/Strong-Secrecy
sh run_check-dps.sh
```

Evaluation result shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-Srq, -Psk-Ri-Ti` and shall be `Observational equivalence is true` for files `-Psk-Siq-Ri-Rr-Re-Eiq-Tr, -Psk -Siq-Rr-Re-Ei-Tr-Ti, -reveal-all-but-Psk`.

Results for PQ-WireGuard⋆. For DeepSec, execute:

```
cd pq_wireguard_star/equivalence_properties/Strong-Secrecy
sh run_check-dps.sh
```

Results shall be `The two processes are not trace equivalent` for all files. For ProVerif, execute:

```
cd pq_wireguard_star/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-Ri, -Psk-Srq` and shall be `Observational equivalence is true` for files `-Psk-Siq-Eiq-Rr-Re, -Siq-Srq-Eiq -Ri-Rr-Re`.

Results for Hybrid-WireGuard. For ProVerif, execute:

```
cd hybrid_wireguard/equivalence_properties/Strong-Secrecy
sh run_check-pv.sh
```

Results shall be `Observational equivalence cannot be proved` for files `-Psk-DH-Eic-Ri, -Psk-DH-Eic-Srq, -Psk-Sic-Eic-Ri, -Psk-Sic -Eic-Srq, -Psk-Src-Ri, -Psk-Src-Srq` and shall be `Observational equivalence is true` for files `-all-dh, -all-pq, -no-reveal, -Sic-Src -Eic-Erc-Siq-Srq-Eiq-Ri-Rr-Re`.

A.5 Notes on Reusability

Our artifacts can be extended to benchmark and evaluate new versions (e.g. fixes) of the protocols we evaluate in our paper.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: A Formal Analysis of Apple's iMessage PQ3 Protocol

Felix Linker
Department of Computer Science, ETH Zurich

Ralf Sasse
Department of Computer Science, ETH Zurich

David Basin
Department of Computer Science, ETH Zurich

A Artifact Appendix

A.1 Abstract

We present the formal verification of Apple's iMessage PQ3, a highly performant, device-to-device messaging protocol offering strong security guarantees even against an adversary with quantum computing capabilities. PQ3 leverages Apple's identity services together with a custom, post-quantum secure initialization phase and afterwards it employs a double ratchet construction in the style of Signal, extended to provide post-quantum, post-compromise security.

We present a detailed formal model of PQ3, a precise specification of its security properties, and machine-checked security proofs using the TAMARIN prover. Our analysis covers both key ratchets, including unbounded loops, which was believed by some to be out of scope of symbolic provers like TAMARIN (it is not!). Our artifact provides a pseudocode specification of PQ3, our model of both PQ3 and its security properties, and case studies illustrating how one can prove protocols like PQ3 using TAMARIN.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Our artifact is available at: https://doi.org/10.5281/zenodo.14710687.

The artifact is structured as follows. We omit files and folders that are only included for technical reasons. Instructions on how to use the scripts are provided in Section A.4.2.

`case-study` Contains the example theories used to illustrate our proof methodology.
`proofs` Contains proofs for lemmas that cannot be proven automatically and a script to check individual proofs.
`spec` Contains an iMessage PQ3 pseudocode specification.
`model.spthy` Our formal model of iMessage PQ3.
`oracle.py` A proof heuristic aiding proof construction.
`prove-auto.sh` A script that constructs proves for all automatically provable lemmas.
`prove-expensive.sh` A script that constructs the proof for an automatically provable but computationally expensive lemma.
`wellformedness.sh` A script that checks that our model is well-formed.

A.2.3 Hardware dependencies

Using Tamarin requires no special hardware. However, some proofs require significant time and memory for construction or verification. Ideally, the machine used should have 140 GB or more of memory. Proofs were constructed and timed on a server with 252 GB of memory and two Intel Xeon E5-2650 v4 CPUs, i.e., a 48-thread server.

However, note that Tamarin uses RAM inefficiently. Tamarin stores the entire proof tree, but a proof only needs to access a path in the proof tree. Therefore, memory compression can help you check proofs on machines with much less memory than indicated. For example, some proofs were constructed on a MacBook Pro with only 32 GB of memory and an Apple M2 Max CPU, where we observed 100 GB of virtual memory usage but only 5-10 GB of physical memory usage.

A.2.4 Software dependencies

All major operating systems can be used. We used Ubuntu 24.04 when timing proofs (see Section A.4.2).

To use our artifact, you must have the following tools installed:

- Tamarin v1.8.0 or higher. Installation instructions are provided at: https://tamarin-prover.com/manual/master/book/002_installation.html. We recommend using `brew`.
- Maude v3.1 or higher. When installing Tamarin with `brew`, you will also install Maude. In any other case, the Tamarin manual provides more information on how to install Maude.

- Python 3. Python 3 is a major, well-supported programming language. We therefore do not provide instructions on how to install it.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Download our artifact and extract it.

A.3.2 Basic Test

Open a shell in the artifacts directory and run the following script:

```
./wellformedness.sh
```

This should take no more than 5 minutes. If the command terminated successfully, you will see a list of lemmas (e.g., `Auto_ChainKeySources`) all marked as `analysis incomplete`. Above that, you will see the Tamarin and Maude versions used. Above that, you should see the text "All wellformedness checks were successful."

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Proofs for all security properties and auxiliary lemmas of our PQ3 Messaging Protocol can be constructed or verified.

(C2): Proofs for all lemmas of our case study models can be constructed.

A.4.2 Experiments

For experiments E2 and E3, our artifact lists approximate time and memory requirements for each script or proof file in the respective README. As noted in A.2.3, you may need much less physical memory than indicated when utilizing your operating system's memory compression.

Experiments (E1)-(E3) together prove iMessage PQ3's security, as described in Section 5 of our paper. (E4) constructs proofs for the case study models described in Appendix A of our paper.

(E1): [Verify Wellformedness] [5 human-minutes + 5 compute-minutes]:
How to: Execute the script `wellformendness.sh`.
Results: Observe that the output contains "All wellformedness checks were successful."

(E2): [Prove auto-provable lemmas] [5 human-minutes]:
How to: Execute the scripts `prove-auto.sh` and `prove-expensive.sh`. Note that the latter script may take a day or more to construct the proof.
Results: Observe that (i) the output of `prove-auto.sh` shows every lemma prefixed with `Auto_*` as "verified", and (ii) the output of `prove-expensive.sh` shows the lemma prefixed with `Expensive_Auto_*` as "verified."

(E3): [Verify constructed proofs] [5 human-minutes]:
How to: Navigate to the `proofs` directory. Execute the script `check.sh` for each `.spthy` file in the `proofs` directory. For example:

```
./check.sh CkCompromise.spthy
```

Results: Observe that when checking every proof file, the lemma named like the proof file is shown as "verified." Also, observe that every lemma in the original file `model.spthy` was shown to be verified in this experiment or experiment E2.

(E4): [Prove case studies] [5 human-minutes + 1 compute-minute]:
How to: Navigate to the `case-study` directory. For each `.spthy` file in the directory, run:

```
tamarin-prover --prove <FILE>
```

Results: Observe that for each file, all lemmas are shown as "verified."

A.5 Notes on Reusability

Our artifact is not intended for reuse. We hope, however, that our case studies serve as educational examples for how to prove protocols secure, which exhibit similar looping behavior to the PQ3 Messaging Protocol.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Towards Practical, End-to-End Formally Verified X.509 Certificate Validators with Verdict

Zhengyao Lin[†] Michael McLoughlin[†] Pratap Singh[†] Rory Brennan-Jones[‡]
Paul Hitchcox[†] Joshua Gancher[§] Bryan Parno[†]
[†]*Carnegie Mellon University* [‡]*University of Rochester* [§]*Northeastern University*

A Artifact Appendix

A.1 Abstract

Paper: Validating X.509 certificates is a critical part of Internet security, but the relevant standards are complex and ambiguous. Most X.509 validators also intentionally deviate from these standards in idiosyncratic ways, often for security or backward compatibility. Unsurprisingly, the result is a long history of security vulnerabilities.

In this work, we present Verdict, the first end-to-end formally verified X.509 certificate validator with customizable validation policies. Verdict's formal guarantees cover certificate parsing, path building, and path validation. To make Verdict practical to both verify and to use, we specify its correctness generically in terms of a user-supplied validation policy written concisely in first-order logic, with a proof-producing compiler to efficient Rust code.

To demonstrate Verdict's expressiveness, we use Verdict's policy framework to implement the X.509 validation policies in Google Chrome, Mozilla Firefox, and OpenSSL, and formally prove that they conform to a subset of RFC requirements. We instantiate Verdict with each policy and show that Verdict matches the corresponding baseline's behavior and state-of-the-art performance on over ten million certificates from Certificate Transparency logs.

Artifact: This artifact includes the source code of Verdict, as well as forks of six other X.509 validators we evaluate against: Chrome, Firefox, OpenSSL, ARMOR, CERES, and Hammurabi. We also provide scripts to automate all three main evaluations we perform in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Verdict is a verified X.509 certificate validator, and as such, it does not pose any immediate security, privacy, or ethical concerns. We also provide a self-contained and compiled version of the artifact as a Docker image, which adds another layer of isolation to avoid changes to your host system.

A.2.2 How to access

The artifact can be accessed via the following URLs:

- Artifact image: https://doi.org/10.5281/zenodo.15468400
- Main Verdict repository: https://github.com/secure-foundations/verdict
- Evaluation tools and build scripts of the artifact image: https://github.com/secure-foundations/verdict-bench

The first Docker image is sufficient for artifact evaluation.

A.2.3 Hardware dependencies

Our artifact image requires an x86-64 machine with at least 16 GiB of RAM and 20 GB of free disk space.

A.2.4 Software dependencies

We recommend using Ubuntu (at least 20.04) with Docker 27.5.1. Newer versions may work too. Your host user must also be able to use the `--cap-add=NET_ADMIN` flag in Docker, in order to perform the end-to-end evaluation with Rustls.

A.2.5 Benchmarks

In the paper, we use about 10M certificate chains from Certificate Transparency (CT) logs. For convenience of evaluation, we only include a sampled subset of 35,000 chains in the artifact image. You are also welcome (but not required) to use your own set of certificate chains, and the instructions to do so are in the `README.md` file in the evaluation tools repository.

All other datasets are included in the artifact image.

A.3 Set-up

From this point onwards, we assume that you are using the pre-build artifact image (Section A.2.2).

A.3.1 Installation

Download the Docker image `verdict-bench-image.tar.gz` (Section A.2.2). Then load the image by

```
docker load --input verdict-bench-image.tar.gz
```

Alternatively, you can also pull the image directly from GitHub for faster access:

```
docker pull \
    ghcr.io/secure-foundations/verdict-bench && \
docker tag \
    ghcr.io/secure-foundations/verdict-bench \
    verdict-bench
```

A.3.2 Basic Test

Start the container by:

```
docker run -it --cap-add=NET_ADMIN verdict-bench
```

Then run `make test` to perform all benchmarks but without multiple samples to reduce noise.

Finally, `make figures` should display a list of LaTeX tables corresponding to Figures 6, 7, and 8 in the paper. The plot for Figure 5 can also be found at `results/performance.pdf`. Note that the performance results may be very noisy and inaccurate at this point, and you should restart the container and run `make eval` to obtain more accurate results (see Section A.4.2).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Verdict's performance is on par with state-of-the-art X.509 validators (Chrome, Firefox, OpenSSL) on a large set of real-world certificates, and it is orders of magnitude faster than prior academic work on more trustworthy X.509 validation (ARMOR, CERES, Hammurabi).*

(C2): *Verdict has high fidelity formal models of Chrome, Firefox, and OpenSSL when compared on the x509-limbo test suite.*

(C3): *When integrated into Rustls, Verdict causes negligible performance overhead in end-to-end HTTPS requests on popular websites.*

(C4): *Verdict is a formally verified X.509 certificate validator with an end-to-end specification.*

A.4.2 Experiments

The following experiments are used to justify the major claims above. In particular, **(E1)** performs benchmarks and tests to demonstrate **(C1)**, **(C2)**, and **(C3)**; **(E2)** builds/verifies Verdict and involves some human inspection to justify **(C4)**.

Experiment **(E1)** does not require Internet connection and all required binaries are included in the Docker image; however, the experiment **(E2)** does need network access to download some dependencies like Rust and Verdict's dependencies.

In all of the following experiments, we assume that the basic set-up steps (Section A.3) have been done, and that you are already inside the artifact container.

(E1) [10 human-minutes + 3.5 compute-hours + 1 GB disk] In this experiment, we perform performance benchmarks and differential tests to demonstrate claims **(C1)**, **(C2)**, and **(C3)**.

How to: Run `make eval`. This might take a few hours.

Results: Run `make figures` to display three LaTeX tables that should have comparable results to Figures 6, 7, and 8 in the main paper in terms of the relative performance of the tools. A PDF file produced at `results/performance.pdf` should also contain similar results to Figure 5 in the paper.

Note that since we are testing on a small subset of CT logs (about 35,000 chains out of the 10 million chains), the differential testing results in the "CT" section will have much fewer certificates; although the "Limbo" section should have similar, and slightly improved results than Figure 7.

Some noise in the performance results (Figures 5, 6, and 8) is also expected (likely within 10% of relative performance between tools), due to the smaller data set as well as differences in the testing environment. However, we expect that the results still support our claim that Verdict is comparable with the performance of Chrome, Firefox, and OpenSSL; and that it is orders of magnitude faster than other academic work.

For more fine-grained results, please see CSV files in the `results` folder:

- `results/perf-<tool>.csv`: performance results of `<tool>` on each tested certificate chain, where the columns denote hash of the leaf certificate, the hostname being validated, validation result, and finally multiple samples of the validation time in microseconds.
- `results/limbo-<tool>.csv`: results of running `<tool>` on the x509-limbo test suite, where the columns denote the test name, the expected result (as indicated by x509-limbo), and the actual result.
- `results/end-to-end-{aws-lc,libcrux}.csv`: performance results of simulating HTTPS requests to popular websites using Rustls with Verdict integrated. The first column is the domain being tested, the second column is the X.509 validator used (`default` for the built-in validator in Rustls,

`verdict-chrome/firefox/openssl` are Verdict models of Chrome, Firefox, and OpenSSL). Rest of the columns are samples of end-to-end request time.

(E2) [30 human-minutes + 5 compute-minutes] In this experiment, we build and verify Verdict using Verus, and manually inspect the specifications and proofs of Verdict. This is to support claim **(C4)**.

How to: Run `make build-verdict`. Note that this requires network access. This command will install Verus and various dependencies, and then verify Verdict using Verus.

Results: If everything is successful, Verus has verified all proofs in Verdict, and produced the final binary at `verdict/target/debug/verdict`.

To manually inspect the specifications and proofs in Verdict, you can find the main components in these locations:

- `verdict` is the main verified X.509 validation library. It includes implementations of different policies (`verdict/src/policy`) as well as the policy-independent validation procedure (`verdict/src/validator.rs`). In particular, the main specification in Figure 2 of the paper can be found in `verdict/src/validator.rs`, where `spec_validate_x509_base64` and `Query::valid` should be equivalent to the `spec_valid_x509` spec function in the paper.
- Verdict's formal models of Chrome, Firefox, and OpenSSL X.509 policies can be found in `verdict/src/policy`. For example, `verdict/src/policy/openssl.rs` contains the OpenSSL policy written in Verdict's policy DSL.
- `verdict-parser` contains the verified parsers and serializers of X.509 (`verdict-parser/src/x509`) and various ASN.1 components (`verdict-parser/src/asn1`). The main paper mentions an ASN.1 DSL to automatically generate verified parsers and serializers, and an example of this can be found in `verdict-parser/src/x509/tbs_cert.rs` (i.e., the `asn1!` macro invocation).

A.5 Notes on Reusability

The Docker image came bundled with a built Verdict frontend binary, which you can find at `verdict/target/release/verdict`. You can run `verdict/target/release/verdict --help` to see its usage. Currently, the frontend supports parsing X.509 certificates, validating given certificate chains, and various utilities to run the evaluation in the main paper.

As an example, `verdict/verdict/tests/chains` contains some sample certificate chains. You can verify one of them by running:

```
verdict/target/release/verdict validate \
    -t 1735707600 \
    chrome \
    verdict/verdict/tests/roots.pem \
    verdict/verdict/tests/chains/github.pem
```

This should produce output such as:

```
result: ValidationResult { valid: true, ... }
```

The Verdict library can also be easily used in other Rust projects through Cargo:

```
cargo add verdict \
  --git https://github.com/secure-foundations/verdict.git
```

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: PICACHV: Formally Verified Data Use Policy Enforcement for Secure Data Analytics

Haobin Hiroki Chen
Indiana University Bloomington

Hongbo Chen
Indiana University Bloomington

Mingshen Sun
Independent Researcher

Chenghong Wang
Indiana University Bloomington

XiaoFeng Wang
Indiana University Bloomington

A Artifact Appendix

A.1 Abstract

Ensuring the proper use of sensitive data in analytics under complex privacy policies is an increasingly critical challenge. Many existing approaches lack portability, verifiability, and scalability across diverse data processing frameworks. We introduce PICACHV, a novel security monitor that automatically enforces data use policies. It works on relational algebra as an abstraction for program semantics, enabling policy enforcement on query plans generated by programs during execution. This approach simplifies analysis across diverse analytical operations and supports various front-end query languages. By formalizing both data use policies and relational algebra semantics in Coq, we prove that PICACHV correctly enforces policies. PICACHV also leverages Trusted Execution Environments (TEEs) to enhance trust in runtime, providing provable policy compliance to stakeholders that the analytical tasks comply with their data use policies.

This artifact contains the Coq code, Rust implementation of PICACHV, and several tools for benchmarking PICACHV.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Since PICACHV is designed to enforce data use policies, it inherently does not pose any apparent security, privacy, or ethical concerns. Moreover, all code, scripts, and documentation are fully transparent and subject to human inspection.

A.2.2 How to access

Our artifact can be accessed on GitHub (evolving) and Zenodo (permanent archive).

- **GitHub:** An evolving version at https://github.com/hiroki-chen/picachv.
- **Zenodo:** A permanent archive: https://zenodo.org/records/14639575

Special Note: A more detailed documentation for performing artifact evaluation can be found in our repository at `docs/ae-readme.md`.

A.2.3 Hardware dependencies

We recommend conducting the evaluation on a high-end server equipped with an x86 server-grade Intel CPU (e.g., 5th Gen Intel Xeon) with Intel Trusted Domain Extension (TDX) enabled and at least 64GB of RAM. While PICACHV does not explicitly depend on TDX for performance evaluation, this hardware feature ensures end-to-end security guarantees.

A.2.4 Software dependencies

We recommend running PICACHV on Ubuntu 24.04 for optimal compatibility. Additionally, we provide a Docker image for seamless testing and evaluation.

For those who wish to build the entire artifact from scratch, the following packages must be installed beforehand.

- **System-wide dependencies:**
```
sudo apt install -y build-essential clang
python3 python3-pip python3-as-python
libgmp-dev pkg-config clang protobuf-compiler
```
- **Rust toolchain:**
```
curl -proto '=https' -tlsv1.2 -sSf
https://sh.rustup.rs | sh -s
```
- **Install `opam` to get Coq:**
```
bash -c "sh <(curl -fsSL
https://opam.ocaml.org/install.sh)"
opam init
eval $(opam env)
opam pin add coq 8.19.0
```

- **(optional) Install `mold`:**
  ```
  cd /tmp
  git clone -branch stable https://github.com/rui314/mold.git
  cd mold
  sudo ./install-build-deps.sh
  cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_COMPILER=c++ -B build
  cmake -build build -j‘nproc‘
  sudo cmake -build build -target install
  ```

- **Install python dependencies for benchmarks and data preparations.**
  ```
  pip3 install polars coloredlogs pydantic pydantic_settings pyarrow
  ```

If you are using docker, please install docker toolchains via docker's official installation guide at https://docs.docker.com/engine/install/ubuntu/

A.2.5 Benchmarks

Our benchmarks primarily require two datasets: (1) a simulated dataset generated using `TPC-H`, which is included in our artifact, and (2) case study data collected from Kaggle, available in a separate repository attached as a submodule in the main repository.

A.3 Set-up

First of all, clone our repository via `git clone --recursive https://github.com/hiroki-chen/picachv.git`

- For docker users:
  ```
  cd picachv
  docker build -f docker/Dockerfile -t picachv/picachv .
  ```
- There is currently nothing to do for users who want to build the artifact from scratch.

A.3.1 Installation

We have already introduced all the installation steps in Section A.2.4.

A.3.2 Basic Test

- **Test if Coq works:** This can be done by simply running the following script in the `picachv-proof-lib` directory of our artifact.
  `./run --allow-admitted` should yield no type errors.
- **Test if `picachv` works:**
  `cd ./benchmarks/micro/polars && cargo b -r` should yield no compilation errors.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The Coq formalism and security proofs are indeed mechanically checked.*
(C2): *The overall runtime overhead is practical compared to insecure baslines shown in the macrobenchmarks in Figure 15 and Table 2.*
(C3): *The runtime overhead is less relevant to the policy type but rather more impacted by the aggregation and project operators that involve policy transformation.*
(C4): *Our policy is indeed expressive enough and can capture real-world use cases as demonstrated in Section 7.3.*

A.4.2 Experiments

Preparation. In order to perform the following experiments, we need to first generate the data tables and policies needed. In our experiments, we use `tpch-dbgen` located in `benchmark`. For the convenience of testing, we provide a Python wrapper script called `prepare_data.py`.

(E1): *End-to-end latency / macrobenchmark [30 human-minutes + 30 compute-minutes + 10GB disk]:* This experiment showcases the runtime overhead of PICACHV when compared with the non-policy-enforced `tpc-h` queries. We illustrate the steps below.
How to: Generate the data and run the `tpc-h` benchmark.
Preparation: At the beginning please use the data preparation script to generate the required data at scale factor 1.0:
`cd benchmark && python3 ./prepare-data.py --scale-factor=1.0`. Then generate the mock policy files for testing.
```
cd ../tools/policy-generator
cargo run -r -- --output-path ../../data/policies/ --format parquet
```
Execution: Navigate back to `benchmark/polars-tpc` and run the following:

- Insecure baseline: `cargo run -r -- --query=[#num]`
- TPC-H with PICACHV: `cargo run -r -- --query=[#num] --policy-path ../../data/policies --enable-profiling`

Replace #num with the query number in Figure 15. (optional) After the results of all queries are collected, please open `tools/plotting/macro.ipynb` and fill in `baseline_polars` and `policy_polars` the results just obtained from the benchmark to produce Figure 15 shown in our paper.
Results: Should be able to reproduce the results in Figure 15.

(E2): *Microbenchmarks [30 human-minutes + 30 compute-minutes + 5GB disk]:* Microbenchmarks will measure 1) the cost breakdown of each relational operator (Figure 14), and 2) two specific benchmarks on the project and aggregate operators.
How to: Generate the data and run the microbenchmark.
Preparation: Similar to **E1**, please have data generator `prepare_data.py` at hand.
Execution: Due to heavy use of commands involved, please kindly refer to the `docs/AE-README.md` for more detailed instructions.
Results: This microbenchmarks should be able to reproduce Figure 14 and 16.

(E3): *Case studies [30 human-minutes + 30 compute-minutes + 5GB disk]*: This experiment shows that PICACHV can indeed support real-world use cases.
How to: Run the case studies and get the results.
Preparation: None.
Execution: Navigate to the `case_studies` repository and then run each the case study program. Due to heavy use of commands involved, similar to E2, please kindly refer to the `docs/AE-README.md` for more detailed instructions.
Results: Table 2 should be reproduced.

A.5 Notes on Reusability

We provide a brief guide on reusing our artifact for future research. The Coq formalism of relational data structures and semantics can be seamlessly integrated into other Coq projects by simply downloading and importing the relevant files. Additionally, the security proofs serve as reference proof tactics for verifying other security properties, extending beyond those discussed in the paper.

The implementation of PICACHV is a standalone Rust library designed for integration with existing data analytics frameworks. This integration requires appropriate code refactoring to invoke the corresponding PICACHV APIs. Additionally, the runtime APIs can be exposed via foreign function calls for non-Rust environments, though some manual effort may be required.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries

Pratap Singh
Carnegie Mellon University

Joshua Gancher
Northeastern University

Bryan Parno
Carnegie Mellon University

A Artifact Appendix

A.1 Abstract

Paper Cryptographic security protocols, such as TLS or WireGuard, form the foundation of a secure Internet; hence, a long line of research has shown how to formally verify their high-level designs. Unfortunately, these formal guarantees have not yet reached real-world implementations of these protocols, which still rely on testing and ad-hoc manual audits for security and correctness. This gap may be explained, in part, by the substantial performance and/or development overhead imposed by prior efforts to verify implementations.

To make it more practical to deploy verified implementations of security protocols, we present OwlC, the first fully automated, security-preserving compiler for verified, high-performance implementations of security protocols. From a high-level protocol specification proven computationally secure in the Owl language, OwlC emits an efficient, interoperable, side-channel resistant Rust library that is automatically formally verified to be correct.

We produce verified libraries for all previously written Owl protocols, and we also evaluate OwlC on two new verified case studies: WireGuard and Hybrid Public-Key Encryption (HPKE). Our verified implementations interoperate with existing implementations and their performance matches unverified industrial baselines on end-to-end benchmarks.

Artifact This artifact includes the Haskell source code of OwlC, as well as the various case studies described in the paper: 14 toy protocols in Owl, an example of a verified application built on top of an Owl protocol, and our performant, interoperable implementations of WireGuard and HPKE. We also provide a Docker container in which to run the evaluations described in our paper, and scripts to automate those evaluations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

OwlC is a security-preserving compiler for cryptographic protocols in the Owl language; thus, using OwlC does not entail direct security, privacy or ethical concerns. Our artifact does not contain any attacks or exploits, nor is it intended to be destructive in any manner. To provide further isolation to users, we provide a Dockerfile which will build a self-contained Docker image with all dependencies for our evaluation.

A.2.2 How to access

Our artifact is available on Zenodo at `https://doi.org/10.5281/zenodo.15605318`. We provide a zip file `owl-usenix2025-aeval.zip` containing all files required for the evaluation.

A.2.3 Hardware dependencies

Our artifact requires an x86-64 machine with at least 32 GB of RAM and 30 GB of free disk space.

A.2.4 Software dependencies

The artifact has been tested on Ubuntu 20.04 and Docker version 24.0.5. More recent versions, or other operating systems, may work as well. We require Python 3 (version 3.8.10 or newer) and `pip3` on the host device, with the packages `matplotlib`, `numpy`, and `prettytable` available; these will be installed during the Set-up phase (A.3). For (**E2**), we require the following flags to Docker, to set up a WireGuard VPN tunnel between two containers:

```
--cap-add NET_ADMIN --device \
    /dev/net/tun:/dev/net/tun
```

All experiments require Internet access to download Haskell, Rust, or Go dependencies.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

1. Ensure that Docker, Python3, and `pip3` are installed and available on the host system.

2. Download and unzip the artifact archive. All remaining steps should be conducted at the top level of the resulting directory `owl-usenix2025-aeval`.

3. Install required Python dependencies:

   ```
   pip3 install -r requirements.txt
   ```

   If desired, the Python dependencies can be installed in a virtual environment such as `venv`.

4. Build the Docker image:

   ```
   docker build -t owlc-aeval .
   ```

   Note that this may take up to 30 minutes, and requires Internet access to download dependencies. Alternatively, a pre-built Docker image is available on the Docker Hub at https://hub.docker.com/repository/docker/ps1729/owlc-aeval/. To pull the image and tag it as `owlc-aeval`, as expected by the scripts:

   ```
   docker pull ps1729/owlc-aeval
   docker tag ps1729/owlc-aeval owlc-aeval
   ```

While our artifact has been developed and tested for an x86-64 Linux machine, other architectures may work as well. To build the image on Apple Silicon, pass the `--platform linux/amd64` argument to `docker build`, as:

```
docker build --platform=linux/amd64 -t \
    owlc-aeval .
```

Note that this will use Rosetta emulation to run the scripts, which will likely incur large overheads and result in very different performance measurements.

A.3.2 Basic Test

We provide a helper script, `run_owlc.sh`, that runs OwlC on a specified file, then verifies the generated library using Verus. The library is generated in `extraction/src/lib.rs`. As a simple test, run:

```
./run_owlc.sh owl_toy_protocols/dhke.owl
```

This will typecheck, compile, and verify our Diffie-Hellman case study. It should take about 2 minutes the first time (subsequent runs may be faster). The script will print `Done!` at the end if everything was successful; it will also print `verification results:: 105 verified, 0 errors` or similar near the end of the output, confirming that Verus verified the generated code.

A.4 Evaluation workflow

Note that some of our evaluations and claims refer to the extended version of our paper, available on ePrint at https://ia.cr/2025/1092. The extended version contains three extra appendices, including Appendices E and F that contain additional details on our WireGuard benchmarks, but is otherwise identical to our submitted version.

A.4.1 Major Claims

(C1): *OwlC is a security-preserving compiler for protocols in the Owl language, producing Rust libraries verified with Verus to be free of explicit and implicit information leakage.* This claim is evaluated in Figure 13 and Section 10 of our paper.

(C2): *OwlC's Rust protocol libraries for WireGuard and HPKE have performance equivalent to state-of-the-art industrial implementations.* This claim is evaluated in Sections 10.2 and 10.3 of our paper, and Appendices E and F of our extended version.

A.4.2 Experiments

Note: You should run all of the `.sh` scripts **on your host machine**; you don't need to spin up the Docker container in interactive mode to run them. The scripts will themselves spin up containers as necessary to run the experiments. All scripts should be run from the top-level directory `owl-usenix2025-aeval`.

(E1): *[Run OwlC on all examples] [5 human-minutes + 20 compute-minutes]:*
This experiment substantiates claim (**C1**) by running OwlC on all sixteen examples from our paper, then using Verus to verify each of the resulting implementations. It should produce a table similar to Figure 13 in our paper.
How to: Run the following command:

```
./run_owlc_on_all.sh
```

Results: Once complete, the script should print a table with the same structure as Figure 13 in our paper. Note that some of the line counts for Verus may differ slightly due to changes in OwlC, and the verification times may be shorter due to improvements in Verus since the paper was accepted. The raw data will be saved to `run_owlc_on_all.csv`, and a copy of the formatted table will be saved to `run_owlc_on_all.txt`.
If any of the table rows contain ERROR, that indicates that either Owl failed to typecheck the file, OwlC failed to compile it, or Verus failed to verify the generated code.

(E2): *[Benchmark WireGuard end-to-end performance] [10 human-minutes + 4 compute-hours]:*
This experiment substantiates claim (**C2**) by measuring the end-to-end performance of several WireGuard implementations based on OwlC's generated WireGuard library, compared against industrial baselines. For details on how we constructed our OwlC-based WireGuard implementations, see Section 9.2.2, and for details on the baselines and comparisons we performed, see Section 10.2.1 and Appendix E in our extended version.
How to: Run the following command:

```
./bench_wireguard_end_to_end.sh
```

This script calls the Python scripts `bench_wg_linedelay.py` and `bench_wg_mss.py` with arguments to run all benchmarks. Since this benchmark needs to orchestrate two Docker containers connected by a Docker network bridge, we run the Python scripts on the host machine.

Results: Once complete, this script will generate four graphs in both `.png` and `.pdf` format, as follows:

- `bench_wg_linedelay-go.{png|pdf}`: Should be comparable to Figure 14 in our paper.
- `bench_wg_linedelay-rs.{png|pdf}`: Should be comparable to Figure 17 in our extended version.
- `bench_wg_mss-go.{png|pdf}`: Should be comparable to Figure 16 in our extended version.
- `bench_wg_mss-rs.{png|pdf}`: Should be comparable to Figure 18 in our extended version.

In all cases, the generated graphs should be comparable in terms of the relative performance of the three WireGuard implementations in question; the precise measured throughput values may differ from our paper due to different hardware and Docker overhead. The raw data for each graph is saved in the corresponding `.csv` files, and in formatted tables in the corresponding `.txt` files.

(E3): *[Benchmark HPKE end-to-end performance] [10 human-minutes + 5 compute-minutes]:* This experiment substantiates claim (**C2**) by measuring the end-to-end performance of our HPKE implementation based on OwlC's generated HPKE library, compared against an industrial baseline. For details on how we constructed our OwlC-based HPKE implementation, see Section 9.3.2, and for details on the baselines and comparisons we performed, see Section 10.3.

How to: Run the following command:

```
./bench_hpke.sh
```

Results: Once complete, this script will generate two text tables, printed to the shell and saved to `bench_hpke.txt`. The raw data will be saved to `bench_hpke.csv`. The contents of the tables should be comparable to Figure 15 in our paper in terms of relative performance; in particular, the percentage differences from the baselines in each table should be comparable.

(E4): *[Micro-benchmark WireGuard handshake] [10 human-minutes + 5 compute-minutes]:*
This experiment substantiates claim (**C2**) by measuring the performance of OwlC's generated routines for the WireGuard handshake in isolation. For discussion of the micro-benchmark, see Section 10.2.2, and for details, see Appendix F in our extended version.

How to: Run the following command:

```
./microbench_wg_handshake.sh
```

Results: Once complete, this script will generate two text tables, printed to the shell and saved to `microbench_wg_handshake.txt`. The raw data will be saved to `microbench_wg_handshake.csv`. The contents of the tables should be comparable to Figure 19 in our extended version in terms of relative performance; in particular, the percentage differences from the baselines in each table should be comparable.

(E5): *[Micro-benchmark WireGuard transport] [10 human-minutes + 45 compute-minutes]:*
This experiment substantiates claim (**C2**) by measuring the performance of OwlC's generated routines for the WireGuard transport layer in isolation. For discussion of the micro-benchmark, see Section 10.2.2, and for details, see Appendix F in our extended version.

How to: Run the following command:

```
./microbench_wg_transport.sh
```

Results: Once complete, this script will generate a graph in `microbench_wg_transport.{pdf|png}`. The raw data will be saved in `microbench_wg_transport.csv`, and in a formatted table in `microbench_wg_transport.txt`. This graph should be comparable to Figure 20 in our extended version in terms of the relative performance of the three routines under test. Note that the generated graph will not have the data series for $\text{OwlC}_{\text{unopt}}$: that corresponds to an old, unoptimized version of OwlC, about which we do not make any performance claims.

A.5 Notes on Reusability

The `run_owlc.sh` script provides a convenient way to run Owl and OwlC on a protocol. OwlC generates a library crate in `extraction/`, with the generated code in `extraction/src/lib.rs`. The other files in `extraction/src/` are handwritten Verus definitions for the generated code; of particular interest may be `speclib.rs`, which contains our definitions of the `ITree` (Section 4.1), `DeclassifyingOp` (Section 5.2), and corresponding token structures (Sections 4.2 and 5.2). In `execlib.rs`, `mod secret` defines our `SecretBuf` abstraction (Section 5.1).

OwlC is under active development as part of the Owl project. The open-source repository is at https://github.com/secure-foundations/owl.

A.5.1 Echo Server case study

As discussed in Section 7, we developed a simple case study of a verified echo server, available in the `echo_server_example` directory. The directory `echo_server_example/extraction` contains a copy of the generated protocol library, with the addition of `server.rs`, a handwritten file containing a verified Verus implementation of the main echo server logic (Section 7.2). To verify the echo server example code with Verus, run the script

`verify_echo_server_example.sh` on your host machine (not inside the Docker container).

The echo server illustrates how to connect a library generated by OwlC to a larger verified codebase, by hand-writing ITree specifications for the application logic. Verus will verify that the application logic follows its handwritten ITree specification. However, as discussed in Section 7, Verus does not presently feature information-flow control or cryptographic reasoning, so it cannot prove cryptographic security properties about handwritten ITrees. Developers must therefore take care that any handwritten ITrees do not leak protocol secrets or otherwise compromise the security properties of the code proven in Owl. The best practice for developing verified applications using OwlC is to write as much of the application logic as possible in Owl, with handwritten ITrees only for simple interfaces and functionality that does not directly interact with secrets.

A.5.2 Incorporating code generated by OwlC into larger Rust projects

OwlC generates a library crate that can straightforwardly be integrated into verified code in Verus. Examples of this are in the `echo_server_example` directory, as we discuss below. Our WireGuard and HPKE case studies illustrate how to integrate the generated libraries into pre-existing Rust codebases. We discuss some high-level considerations here.

Integrating a library generated by OwlC into a pre-existing Rust codebase requires building an interface between verified and unverified code. As with any such interface, the developer must make sure that the unverified calling context satisfies the preconditions of the verified routine. For OwlC, this means that all arguments to generated protocol routines must match the Owl types of the corresponding Owl protocol routine. That is, if an Owl routine takes as argument a key k, then the generated Verus routine must only be called with the runtime value of the key k.

OwlC relies on Rust abstract types and Verus verification conditions to maintain important soundness and correctness properties. Our ITree tokens (Section 4.2) cannot be created or duplicated, so that implementation code cannot forge tokens to justify incorrect I/O operations. Our `SecretBuf` wrapper types (Section 5.1) are opaque, so that implementation code cannot leak secrets via side channels. However, connecting to unverified Rust code may necessitate breaking some of these abstraction barriers. For instance, an application may need to reveal a `SecretBuf` to display a decrypted plaintext message to the user.

When connecting to unverified code, developers may therefore need to add "escape hatches" that enable such functionality. This must be done with extreme care. We recommend that any such escape hatches be annotated with a Verus precondition of `requires false`, so that verified code cannot use them, and the generated libraries remain sound.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Vulnerability of Text-Matching in ML/AI Conference Reviewer Assignments to Collusions

Jhih-Yi (Janet) Hsieh, Aditi Raghunathan, Nihar B. Shah
School of Computer Science, Carnegie Mellon University
jhihyih@alumni.cmu.edu, {aditirag, nihars}@andrew.cmu.edu

A Artifact Appendix

A.1 Abstract

In addition to releasing all adversarial abstracts generated in this work, our artifacts include the code, datasets, and LLM prompts necessary for researchers to reproduce the *fully automatic* mode attack results that support the paper's main claims. In this artifact appendix, we describe in detail how each of our main results can be reproduced and verified.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There is no risk for evaluators while executing our artifacts. However, it is important to remember that any successful attack pertains to the abstract and/or reviewer profile modified in this experiment and not to the original abstract or profile in the conference. Finally, do not use this artifact to produce adversarial abstracts for your submissions. Safeguards against this attack are implemented at major AI/ML conferences.

A.2.2 How to access

Our research artifacts are available on Zenodo `https://doi.org/10.5281/zenodo.15588237`.

A.2.3 Hardware dependencies

We have tested the reproduction instructions in this document on CPU-only devices as well as GPU-enabled. A GPU can provide substantial speed-ups. The hardware needs to be compatible with the PyTorch deep learning framework. The minimal hardware the artifacts have been tested on has an 8 CPU cores with 16 GB RAM, but having more is helpful. If you do not have GPU access but wish to gain one, we have tested on and recommend AWS (instance type: g4dn.xlarge, image: Deep Learning OSS Nvidia Driver AMI GPU PyTorch 2.7 (Ubuntu 22.04)). Your machine should have at least 20 GB of free memory for the datasets, virtual environment, and models.

A.2.4 Software dependencies

We package our environment using Conda virtual environments. We recommend installing Miniconda, which is a minimal installer of Anaconda. Please follow the instructions on their website: `https://www.anaconda.com/docs/getting-started/miniconda/install`.

Our code makes calls to the OpenAI API (paid service), so an API key with available credits is needed.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

First, download the `dataset.zip` file from Zenodo (`https://doi.org/10.5281/zenodo.15588237`) and extract the files locally. There should be a `datasets/` directory containing both NeurIPS 2022 and 2023 datasets. You may rename the outer directory `datasets/`, but please keep everything else the same way.

Now find `reviewer_assignments_vulnerability.zip` from the same Zenodo link and extract it somewhere outside of the datasets directory. This directory contains the code, adversarial abstract examples, LLM prompts, and other scripts. Change into the `reviewer_assignments_vulnerability` directory, and create the virtual environment using the following:

```
$ conda env create -f environment.yml
```

Activate the environment:

```
$ conda activate attack
```

Set the OpenAI API key environment variable.

```
$ export OPENAI_API_KEY=your_key
```

A.3.2 Basic Test

The basic test below simulates the proposed collusion attack between one paper-reviewer evaluation pair. Set the `--datasets_dir` flag to the directory path that contains both NeurIPS 2022 and 2023 datasets (for example `"../datasets"`). Run the command below from the code repository. If you have access to a GPU, first test if GPU is availabe for PyTorch (try `torch.cuda.is_available()` in Python console) and then add the flag `--device=cuda` to the command below. *You must specify the device or it will default to running on CPU only.*

```
(attack) $ python attack/main.py \
--datasets_dir=path/to/datasets/directory \
--end=1
# Optional arguemnt: --device=cuda
```

After the run successfully finish, you should find an output directory `out/yyyy-mm-dd/hh-mm-ss` under the code repository. It will contain a sub-directory called `attack_0`, under which you will find the attack logs and information. Open `attack_info.json` and you will find the attack metadata and results, including the adversarial reviewer archive and the modified abstract.

To evaluate the run output, you should use the following script. However, for the basic test, you may see NaN and errors since there is only one sample (for example, variance and confidence intervals are meaningless).

```
(attack) $ python scripts/run_result.py \
--run_dir=out/yyyy-mm-dd/hh-mm-ss
```

A.4 Evaluation workflow

This section outlines the necessary procedures and experiments required to verify artifact functions and repoducibility of the major claims in the paper.

A.4.1 Major Claims

Here we enumerate the major claims made in the paper.

(C1): The proposed attack procedure successfully manipulates reviewer assignment. This is proven by experiment (E1) described in Section 5.3 whose results are reported in Table 2.

(C2): Policies that require reviewers to have more papers in their archives can reduce the effectiveness of the proposed attack. This is proven by experiment (E2) described in Section 5.4 whose results are illustrated in Figure 4.

(C3): To combine the similarity scores between a submitted paper and a reviewer's multiple papers in their archives, using average pooling instead of max pooling can help reduce the impact of targeted manipulations, making the proposed attack less successful under average pooling. This is proven by experiment (E3) described in Section 5.5 whose results are reported in Table 4.

(C4): Considering NeurIPS 2022 as a "past" conference whose data is publicly available to an attacker, and NeurIPS 2023 as the "current" conference whose data is unavailable, we find that the attack performance on NeurIPS 2022 is reflective of the performance in 2023. This is proven by experiment (E4) described in Section 5.6 whose results are illustrated in Figure 5.

A.4.2 Experiments

For each major claim above, we provide the instructions to reproduce the experiment in our paper that supports the claim. The instructions here reproduce these experiments at a smaller scale in terms of both sample size and the number of scenarios, but we believe thst they are representative and sufficient to support the main claims. Due to the stochastic nature of LLM-generated abstract attacks and the low sample sizes, we do not expect the result numbers to be exactly the same as reported in the paper, but we expect them to still support each claim.

(E1): *[Attack Success Rates] [20 human-minutes + 3 compute-hour]: run the attack procedure on paper-reviewer pairs with natural rankings 101 and 1001.*

Preparation: Make sure your OpenAI API key has sufficient credit. We estimate about 5 to 10 dollars for this experiment.

Execution: Use the following command to execute the attack procedure for 20 samples. You will need to run this command twice (one after the other has completed), with different input argument to `--samples_file`, for natural rankings of 101 and 1001 (replace the `attacks_rank101.jsonl` file with `attacks_rank1001.jsonl` below). Two runs take about 3 hours total on CPU and 2 hours on GPU.

```
(attack) $ python attack/main.py \
--datasets_dir=path/to/datasets/directory \
--samples_file="evaluation_samples/\
attacks_rank101.jsonl" \
--end=20
# Optional arguemnt: --device=cuda
```

Results: Each run (each time `attack/main.py` is executed) will have its own output directory, named by the timestamp of when the run started. Use the evaluation script introduced at the end of Appendix A.3.2 to evaluate the attack results of each run. Compare the attack success rates and manipulated rankings statistics with the first and last rows of Table 2 in Section 5.3. Since the sample size of 20 here is much lower than in Table 2, higher variance and wider confidence intervals are expected.

(E2): *[Reviewer Archive Length] [15 human-minutes + 4 compute-hour]: run the attack procedure, but reviewers*

have to keep 10 papers (instead of 1) in their archives.

Preparation: Make sure your OpenAI API key has sufficient credit. We estimate about 10 dollars for this experiment.

Execution: Use the following command to execute the attack procedure for 20 samples. This will take about 4 hours total on CPU and about 1.5 hours on GPU.

```
(attack) $ python attack/main.py \
--datasets_dir=path/to/datasets/directory \
--samples_file="evaluation_samples/\
arclen_attacks_rank101.jsonl" \
--num_papers_to_keep=10 --end=20
# Optional arguemnt: --device=cuda
```

Results: Again, an output directory named by the timestamp of when the run started will be produced. Use the evaluation script introduced at the end of Appendix A.3.2 to evaluate the attack result. Compare with the right-most data point in Figure 4. You should see that the attack success rates are low when reviewers keep 10 papers in their archive.

(E3): *[Average vs Max] [30 human-minutes + 6 compute-hour]: run the attack procedure under average (mean) versus max pooling methods to combine the similarities to each of the reviewer's papers (when reviewers must keep all their past papers).*

Preparation: Make sure your OpenAI API key has sufficient credit. We estimate about 10 dollars for this experiment.

Execution: Use the following command to execute the attack procedure for 20 samples under max pooling. This will take about 2 hours total on CPU and about 1 hour on GPU.

```
(attack) $ python attack/main.py \
  --datasets_dir=path/to/datasets/directory \
  --similarity_mode="max" \
  --samples_file="evaluation_samples/\
  maxsim_attacks_rank101.jsonl" \
  --num_papers_to_keep=10 --end=20
# Optional arguemnt: --device=cuda
```

Next, use the following command to execute the attack procedure for 20 samples under average (mean) pooling. This will take about 4 hours total on CPU and about 1.5 hours on GPU.

```
(attack) $ python attack/main.py \
--datasets_dir=path/to/datasets/directory \
--similarity_mode="avg" \
--samples_file="evaluation_samples/\
attacks_rank101.jsonl" \
--num_papers_to_keep=10 --end=20
# Optional arguemnt: --device=cuda
```

Results: Each run (each time `attack/main.py` is executed) will have its own output directory, named by the timestamp of when the run started. Use the evaluation script introduced at the end of Appendix A.3.2 to evaluate the attack results of each run. Compare with the two rows of Table 4 in Section 5.5. Since the sample size of 20 here is much lower than in Table 4, higher variance and wider confidence intervals are expected. If the results do not show a convincing difference between average and max pooling, we recommend increasing the sample size to 40 for both mean and max scenarios. You can do so by setting the arguments `--start=20` and `--end=40`. In the new run output folders, you will find subdirectories `attack_i/` for $i = 20...39$. To evaluate all 40 samples, you can move those subdirectories into the same run folder as the first 20 samples and run the evaluation script described at the end of Appendix A.3.2 with the run directory that contains all 40 subdirectories.

(E4): *[Correlation] [10 human-minutes + 0 compute-hour]: calculate the correlation of manipulated rankings amongst NeurIPS 2022 and 2023 reviewers.*

Preparation: You must complete (E1) before this experiment.

Execution: Use the following command to calculate the Spearman's rank correlation coefficients between manipulated rankings in NeurIPS 2022 and 2023. Run the follow command twice, setting `--run_dir` to the output directories for natural rankings 101 and 1001 runs from (E1).

```
(attack) $ python scripts/run_result.py \
--run_dir "out/yyyy-mm-dd/hh-mm-ss" \
--correlation
```

Results: The correlation should be high. Compare with the Spearman's rank correlation coefficients reported in Figure 5 of the paper.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: SoK: Machine Learning for Misinformation Detection

Madelyne Xiao
Princeton University

Jonathan Mayer
Princeton University

A Artifact Appendix

A.1 Abstract

We survey literature on automated detection of misinformation across a corpus of 248 well-cited papers in the field. We then examine subsets of papers for data and code availability, design missteps, reproducibility, and generalizability. Our paper corpus includes published work in security, natural language processing, and computational social science. We demonstrate the limitations of current detection methods in a series of three representative replication studies: these replications can be found in sections 5.1, 5.2, and 5.3 of the paper, and address article-, user-, and website-scoped misinformation detection, respectively. The paper artifact contains code and datasets for reproducing the results found in these replication sections

A.2 Description & Requirements

No special setup required beyond access to recent versions of Python (3.10 or later) and IPython (7.16 or later). All Python library requirements are specified in the accompanying `requirements.txt` file in the root directory.

A.2.1 Security, privacy, and ethical concerns

We anticipate no risks to evaluators as a result of code execution.

A.2.2 How to access

For purposes of evaluation, code and data are available at https://github.com/citp/sok_misinformation and https://zenodo.org/records/15613696.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Python 3.10 (or later) required to run pyscripts and a Jupyter notebook. All Python library requirements are specified in the accompanying `requirements.txt` file in the root directory.

A.2.5 Benchmarks

Datasets required for each test are described within the relevant directory in the replications repository. These include the FAKES and ISOT datasets used in the article-level replication, the troll and non-troll datasets used in the user-level replication, and the EMNLP datasets used in the website-level replication. All are included in the replications archive.

A.3 Set-up

No special setup required beyond access to recent versions of Python (3.10 or later) and IPython (7.16 or later).

A.3.1 Installation

Evaluators working from the Git repository should additionally download the `articles_data.zip` archive linked in the git repository; package requirements are included in each pyscript where necessary.

A.3.2 Basic Test

Evaluator may run `python CNN_revised.py -test_source nytimes -rand_seed 0` within the `articles` directory to observe evaluation of a small test dataset in advance of completing all 30 test cases. Output will include accuracy and error reporting.

To observe the evaluation of a small test dataset used in the website-level replication tests, evaluator may run `python3 train_artifact.py -tk fact -f articles_body_glove,articles_title_glove -sb small` within the `websites` directory. Output will include accuracy and error reporting.

A.4 Evaluation workflow

We include summaries of high-level results in the following subsection; detailed discussion of these points can be found in section 5 of the manuscript.

A.4.1 Major Claims

(C1): Article-level detection methods are susceptible to text dependencies resulting from authorship, style, and

source. We demonstrate this through a series of analyses on datasets whose credible/factual texts are sourced from a reputable news agency (Reuters). We find that altering the semantic content of news articles without altering the style of those articles induces no change in the ultimate classification of those articles, suggesting that there is 1) no semantic understanding of news article contents during classification, and 2) style signals likely override any other semantic signals.

(C2): User-scoped detection tasks are frequently tautological, and proceed from known lists of troll users. These classifiers can only detect users who very closely resemble users who were manually identified as trolls, oftentimes using data not included in available training sets.

(C3): Website-scoped detection tasks purport to leverage multimodal feature sets—including infrastructural, textual, and social media data—but the performance of these methods is often overdetermined by features derived from website text semantics (and, as such, the classifier we evaluate is approximately as accurate as a text-only classifier, and is susceptible to the same dependencies we discuss in C1).

A.4.2 Experiments

Articles. See Table 2 for results of a replication analyses of the ISOT, FA-KES, and NYTimes and Reuters datasets. The original paper, which presents results on the ISOT and FA-KES datasets, can be found here: https://www.researchgate.net/publication/348379370_Fake_news_detection_A_hybrid_CNN-RNN_based_deep_learning_approach.

In the `articles` directory of the main replications repository, execute `./run.sh` to run 30 iterations of the model (with prespecified random seeds) on 1) ISOT and FAKES datasets used in the original paper and 2) original and modified datasets we developed for purposes of robustness testing. With multithreading (four cores, each running a separate instance of the model with the same seed and a different dataset), this script runs in about 7.5 hours.

Users. See Figure 4 for the results of a partial dependence analysis on TrollMagnifier data. The original paper can be found here: https://arxiv.org/pdf/2112.00443. The Jupyter notebook containing code to train and run the classifier and create these PDP plots is `trollmagnifier_pdp_artifact.ipynb`. Instructions for use are included in the notebook.

Websites. See Table 4 in Appendix D for the results of ablation analyses on an EMNLP dataset for this paper: https://aclanthology.org/D18-1389.pdf. To run all tests, execute the bashscript contained in the `websites` directory using `./run.sh`.

A text header appears at the start of each test case (e.g. "fact, full corpus, all features") denoting 1) the classification task (factuality or bias classification), 2) the corpus to be used (these vary by size (full, medium, small), or are stratified by bias (left, center, right), or credibility (low, mixed, high)), and 3) the feature set. Feature sets are printed to the console at runtime. Extraction code for features is included in the features directory. You can additionally inspect feature sets and function calls in the 'run.sh' bashscript.

Accuracy, MAE, and F1 scores are reported for each test. To compare these results to those found in Table 4, read each table row horizontally, matching accuracy scores to those reported on the console. Hold-one-out analyses are denoted by (-) in the feature set labels (e.g., "articles(-)" denotes *all* features excluding those relating to article contents and title). Evaluation of specific feature sets are denoted by (+) (e.g., "articles" denotes only those features derived from article contents and title). With the same hardware specs reported previously, the full script runs in about 70 minutes.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: CoreCrisis: Threat-Guided and Context-Aware Iterative Learning and Fuzzing of 5G Core Networks

Yilu Dong, Tianchang Yang, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Ali Ranjbar,
Kai Tu, Tianwei Wu, Md Sultan Mahmud, Syed Rafiul Hussain
The Pennsylvania State University
{yiludong, tzy5088, abdullah.ishtiaq, szr5848, aranjbar, kjt5562, tvw5452, mqm7099, hussain1}@psu.edu

A Artifact Appendix

A.1 Abstract

We present CoreCrisis, a context-aware black-box testing framework for 5G Core Network (5GC) implementations. Considering the stateful nature of network protocols, CoreCrisis adopts a state-aware strategy to navigate the implementation effectively. Unlike previous security analysis efforts of cellular networks which rely on manually-crafted, static testing symbols and are limited to identifying only logical errors, CoreCrisis implements a dynamic two-step approach. Initially, CoreCrisis builds an initial finite state machine (FSM) representation of the 5GC's implementation using only benign symbols (expected inputs) with its novel divide-and-conquer and property-driven equivalence-checking learning. During testing, it utilizes the learned FSM to target under-explored states and introduces attacking symbols (mutated inputs). Based on the responses observed from the Core Network, CoreCrisis continuously refines the FSM to better guide its exploration. Evaluating CoreCrisis on four 5GC systems, including both commercial and open-source implementations, we identified 7 categories of deviations from the technical specifications and 13 crashing vulnerabilities. These logical and crashing vulnerabilities could lead to denial-of-service (DoS), authentication bypass, and billing fraud.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Our artifact is available at Zenodo: https://zenodo.org/records/15043603. We also hosted on GitHub: https://github.com/SyNSec-den/CoreCrisis.

A.2.3 Hardware dependencies

Our artifact does not have specific hardware requirements. However, if the processor is not fast enough or the system does not have enough memory, the artifact may run into issues. The experiment **E1** requires the system to have at least 32GB of memory.

A.2.4 Software dependencies

We tested the artifact on Ubuntu >= 20.04, other Linux distributions may also work.

To run the experiments, docker, JDK11, GCC, and python3 must be installed in the system.

Detailed instructions to install these dependencies are provided inside the artifact.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

First, the required files should be downloaded from Zenodo or GitHub. Then, follow the readme file in the main folder and each subfolder to build and run the program.

A.3.2 Basic Test

A basic functional test can be conducted by running the CoreFuzzer. By following the readme files, the program should successfully run in a docker container with terminal outputs.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The property-driven equivalence checking reduced the number of queries needed to learn the state machine, described in Section 6.3.1 in the paper. We show this in (E1).

(C2): CORECRISIS has been used to identify new bugs in 5G core network implementations, as shown in Table 2 in the paper. This can be proven by (E2).

(C3): We compared CORECRISIS against other fuzzers, as shown in Section 6.2 in the paper. This can be proven by (E3).

A.4.2 Experiments

(E1): [Property-driven equivalence checking] [1 human-hour + 15 compute-hour]:
How to: The difference can be seen by enabling and disabling the Property-driven equivalence checking in state machine learning.
Preparation: Follow the instructions in Corelearner folder to run the program one time.
Execution: We provided the equivalence checking queries under the folder "CEStore". The evaluator should run the state machine learning with the in lines in "input" and check "learner.log" for the number of queries. Then the evaluator should repeat the steps with an empty input file and and check learner.log again.
Results: By checking the logs and the learned state machine, the number of queries should be significantly reduced if Property-driven equivalence checking is enabled.

(E2): [CoreFuzzer] [1 human-hour + 24 compute-hour]:
How to: The evaluator should follow the instructions and run the CoreFuzzer.
Preparation: Follow the instructions and build the docker containers for CoreFuzzer.
Execution: Enter the docker container and run the file "run_hourly.py". The database files used for logging the results should appear in the "logs" folder.
Results: The database file should indicate some crashes from the core network implementation.

(E3): [Comparison with other fuzzers] [1 human-hour + 24 compute-hour]:
How to: The evaluator should follow the instructions and run the CoreFuzzer along with other fuzzers provided for 24 hours to compare the coverage numbers.
Preparation: Follow the instructions and build the docker containers for each fuzzers.
Execution: Enter the docker containers and run the file "run_hourly.py". The coverage information should appear in the "logs" folder.
Results: Our implementation should outperform all the compared implementations.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Gotta Detect 'Em All: Fake Base Station and Multi-Step Attack Detection in Cellular Networks

Kazi Samin Mubasshir*, Imtiaz Karim*, Elisa Bertino
Purdue University

A Artifact Appendix

A.1 Abstract

This artifact provides a machine learning–based toolkit for detecting Fake Base Stations (FBS) and Multi-Step Attacks (MSAs) in cellular networks from network traces in the UE. It includes curated layer-3 cellular network traces (NAS/RRC) along with scripts for preprocessing, model training, evaluation, cross-validation, and performance visualization. Users can easily experiment with a variety of classification approaches—including Random Forest, Support Vector Machines, XGBoost, CNN, LSTM, and Graph Neural Network—by running the provided Python scripts. Metrics such as accuracy, precision, recall, and F1-score are automatically calculated, and visual outputs are generated for further analysis. Beyond classical ML models, the repository also contains scripts for graph-based models (GCN, GAT, GATv2, GraphSAGE, Graph Transformer) and a stateful LSTM with attention. These scripts demonstrate how sequential data (e.g., NAS/RRC messages) and structured data (e.g., graphs of packet traces) can be leveraged for attack detection. Additional scripts (e.g., cross-validation, trace-level classification, and feature extraction) further streamline the ML workflow. While the repository includes our implementation of a signature-based detection approach for comparison, the main focus is on training and evaluating data-driven ML models to showcase their effectiveness in detecting FBSes and MSAs from cellular network traffic.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact does not pose any security concern.

A.2.2 How to access

Datasets and codebase are publicly available at https://zenodo.org/records/14720824 and https://github.com/fbsdetector/fbsdetector-codes.

A.2.3 Hardware dependencies

Minimum Requirements

- **Processor**: Modern multi-core CPU (e.g., 2–4 cores).
- **Memory (RAM)**: 8 GB (sufficient for handling moderate-sized datasets and basic training).
- **Storage**: At least 10 GB of free disk space (for the repository, datasets, logs, and virtual environment).
- **Graphics**: Integrated GPU or CPU-only (sufficient for small-scale or classical ML tasks).

Recommended Requirements

- **Processor**: 4+ cores (e.g., Intel i5/i7 or AMD equivalent).
- **Memory (RAM)**: 16 GB or more (for larger datasets and parallelized model training).
- **Storage**: 20 GB or more of free disk space.
- **GPU (Optional)**: For CNN, LSTM and Graph Neural Networks, an NVIDIA GPU with CUDA support is recommended.

A.2.4 Software dependencies

Operating System

- Tested on Linux (e.g., Ubuntu 22.04 LTS).
- Compatible with Windows and macOS (minor adjustments may be needed).
- Requires a working Python 3.7+ environment.

Core Software Packages

- **Python 3.7+**: Check version with `python3 -V`. We have tested this on `Python 3.8.10`, `Python 3.10.16` and `Python 3.11.5`

* Equal contribution. The student author's name is given first.

- **Pip**: Use `pip install -r requirements.txt` to install dependencies.
- **Tshark 4.4.0**: TShark is the command-line version of Wireshark, a powerful network protocol analyzer. Below are the installation instructions for different operating systems.

  Installation on Linux (Ubuntu/Debian) To install TShark on a Linux system, run the following commands:

```
sudo add-apt-repository ppa:wireshark-
    dev/stable
sudo apt update
sudo apt install wireshark
```

  Check installation

```
tshark --version
```

  If version 4.4.0 is not installed, you can manually download and install it:

```
wget https://www.wireshark.org/download/
    src/all-versions/wireshark-4.4.0.tar.
    xz
tar -xf wireshark-4.4.0.tar.xz
cd wireshark-4.4.0
./configure
make
sudo make install
```

  Installation on macOS (Using Homebrew) To install TShark on macOS, use Homebrew:

```
brew install wireshark
```

  Check installation

```
tshark --version
```

  If an older version is installed, uninstall and install the specific version:

```
brew uninstall wireshark
brew install wireshark@4.4.0
```

  Installation on Windows To install TShark on Windows:

  1. Download Wireshark 4.4.0 from the official website: https://www.wireshark.org/download.html
  2. Run the installer and ensure the **"TShark"** component is selected.
  3. After installation, verify the version by running:

```
tshark --version
```

- **Virtual Environment (Optional)**: Recommended (e.g., `venv` or `conda`).

Key Python Libraries

- **NumPy, pandas, scikit-learn** for data processing and classical ML.
- **PyTorch** and **TensorFlow**.
- **networkx** for graph-based models.

GPU Support (Optional)

- An NVIDIA GPU with CUDA is recommended for large graphs (long traces).
- Ensure compatible CUDA drivers for your PyTorch/TensorFlow version.

Additional Tools (Signature-Based Detection)

- **Graphviz**: For processing `.dot` files (`sudo apt-get install graphviz` on Linux).

A.2.5 Benchmarks

Apart from the datasets provided, PHOENIX's signatures and packet traces are required to test our implementation of PHOENIX for comparison.

A.3 Set-up

A.3.1 Installation

1. Download the Artifact

1. Navigate to the GitHub repository page in your browser.
2. Click on the `Code` button and select `Download ZIP`, or clone via:

```
git clone https://github.com/fbsdetector/
    fbsdetector-codes.git
```

3. After downloading or cloning, change directory into the repository folder:

```
cd fbsdetector-codes
```

2. (Optional) Create and Activate a Virtual Environment

1. Create a virtual environment (recommended to avoid dependency conflicts):

```
python3 -m venv venv
```

2. Activate the environment:

```
source venv/bin/activate
```

3. On Windows, activation is done via:

```
venv\Scripts\activate
```

3. Install Required Dependencies

```
pip install --upgrade pip
pip install -r requirements.txt
```

A.3.2 Basic Test

To verify that your setup is correct and the core functionality works, run:

```
python3 codes/trace-level-classification.py
```

This script performs a trace-level classification using various ML models and should produce output indicating model performance (e.g., accuracy, precision, recall).
Notes

- If you plan to use GPU-accelerated training, ensure your system has the appropriate CUDA drivers installed for *PyTorch* or `TensorFlow`.
- For detailed usage of specific scripts, consult the repository's README or script-level docstrings.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We create two real-world datasets, **FBSAD** and **MSAD**, containing diverse traces of FBSes and MSAs under different scenarios. These datasets are large-scale and high-quality, being the first of their kind to capture FBS/MSA behaviors from real-world cellular network traffic. These datasets are available in the dataset folder.

(C2): We design a two-step detection framework—a packet-level classification followed by a trace-level classification—that outperforms existing solutions in accurately detecting FBSes. Using a novel **stateful LSTM with attention** in the packet-level stage significantly improves detection performance (Experiment (E1), Paper Section 6.1). Results in Tables 2 and 3 confirm this claim.

(C3): We introduce a **graph-based learning approach** for detecting multi-step attacks (MSAs). By representing attack signatures as graphs, Graph Neural Network (GNN) models effectively capture relational patterns even in *unseen* or *reshaped* MSAs. This capability is demonstrated by Experiment (E2), where our method outperforms standard ML baselines and continues to detect evolving MSAs through maximal overlapping subgraphs. Results in Table 5, 16 and 17 confirm this claim.

(C4): We deploy the overall framework in a mobile app and validate its performance in real-world setups. The solution adds a little overhead in terms of memory and power consumption which is demonstrated in Experiment E3. Results can be verified from Figure 4 (a-c) in the paper.

(C5): We implement a functional signature-based implementation of PHOENIX to compare our solution with it. The functionality of the implementation can be verified using Experiment E4. We use the NAS and RRC packet fields as the features of our dataset (reported in Table 15), which can be verified from Experiment E5.

A.4.2 Experiments

Note: The compute time provided is based on running with a GPU, running the experiments with a CPU will take longer.

(E1): *Packet-Level and Trace Level Classification [0 human-minutes + 2-3 compute-minutes + 5GB disk]*
Link to Claims: This experiment demonstrates claims made in **(C2)**, showing the two-step framework's performance at the packet level and trace level.
Preparation: • Confirm that `PyTorch` and `TensorFlow` is installed and GPU drivers are correctly configured if using GPU acceleration.
Execution: • Run:

```
python3 codes/classification-
    models.py dataset/[fbs_nas/
    msa_nas/fbs_rrc/msa_rrc].csv
```

```
python3 codes/stateful-lstm-w-
    attn.py dataset/[fbs_nas/
    fbs_rrc].csv
```

for the packet level classification and

```
python3 codes/trace-level-
    classification.py
```

for trace level classification.

- Wait for the training to finish (time may vary based on hardware, typically 1–2 minutes on a mid-range config).

- The scripts output performance metrics (accuracy, precision, recall, F1-score) for the classification models.

Results:
- Compare performance metrics (F1-score, precision/recall) against Tables 2 and 3.
- We reported the best performance observed across multiple runs with different epochs during training to ensure we capture the most optimal performance of each model. So, results may deviate slightly across runs from the ones reported.

(E2): *Graph-Based MSA Detection [0 human-hour + 3 compute-minutes + 5GB disk]*

Link to Claims: This experiment supports **(C3)**, showcasing how graph-based learning models detect multi-step attacks (MSAs), including those that are previously unseen or reshaped.

Preparation:
- Confirm that `networkx` and the relevant GNN libraries (e.g., PyTorch Geometric and DGL) are installed.

Execution:
- Run the graph-based detection script

```
python3 codes/graph_models.py
    dataset/[msa_nas/msa_rrc].csv
```

- The script will train the specified graph neural network (GNN) models (Graph Convolutional Network, GraphSAGE, GAT, etc.) and output model performance metrics (accuracy, precision, recall, f1-score).
- Run the cross-validation script

```
python3 codes/cross-
    validation.py dataset/[
    msa_nas/msa_rrc].csv
```

- This script performs leave-one-class-out cross-validation and generates the following outputs: (1) Accuracy for each fold, (2) Detailed results for each fold, including true and predicted labels, (3) A pivot table summarizing the true and predicted labels across all folds.

Results:
- Examine the results and verify performance for MSA detection in Table 5.
- Confirm the model's robustness by cross validation (i.e., unseen attack traces) against Table 16 to verify Claim (C3).

(E3): *Real-World Deployment and Comparison [0 human-hours + 1 compute-minute + 5GB disk]*

Link to Claims: Validates **(C4)** by deploying the **FBSDetector** framework in a mobile environment and comparing its performance to signature-based methods.

Execution:
- Run

```
python3 codes/ml-stats.py
```

- This script generates the following plots: (1) Accuracy vs Sequence Length for NAS and RRC datasets (Figure 6 c-d), (2) Time Consumption vs Number of Packets (Figure 4a), (3) Power Consumption vs Number of Packets (Figure 4b) and (4) Memory Consumption vs Number of Packets (Figure 4c).

Results:
- Compare the generated figures against the figures in the paper.

(E4): *Testing PHOENIX implementation for Signature-Based Detection [3 human-minutes + 3 compute-minutes + 2GB disk]*

Link to Claims: This experiment demonstrates the functionality of our signature-based implementation (PHOENIX) as claimed in **C5**, validating its compatibility and accuracy on known attack traces.

Preparation:
- Place the PHOENIX signatures (e.g., `.dot` files for DFA or Mealy machines, PLTL formulas) into the `dataset/signatures/` folder.
- Ensure sample `.pcap` traces with known anomalies are accessible in `dataset/` (or any specified folder).
- Install `graphviz` (for processing `.dot` files) and ensure that `tcpdump` or `Wireshark` is available if needed for additional `.pcap` inspections.

Execution:
- Run the Deterministic Finite Automaton (DFA) test:

```
python3 phoenix-implementation/dfa.py
    dataset/signatures/dfa/NAS/
    attach_reject/attach_reject_50_40.
    trace.dot dataset/NAS_PCAP_logs/
    attach_reject.pcap
```

- Run the Mealy Machine (MM) test:

```
python3 phoenix-implementation/mm.py
    dataset/NAS_PCAP_logs/attach_reject.
    pcap
```

- Run the PLTL test:

```
python3 phoenix-implementation/pltl.py
    dataset/NAS_PCAP_logs/attach_reject.
    pcap
```

- Each script processes the given trace, applies the signatures, and reports detections/anomalies in the console output.

Results:
- Confirm that PHOENIX flags known anomalies (e.g., `attach_reject`) in the sample `.pcap` traces.

- Inspect the console logs for any parsing or detection errors. Successful detection of the known malicious patterns (e.g., attach reject events) validates the correctness of the PHOENIX implementation.

(E5): *[Additional] Comparing Extracted Feature Names [0 human-minutes + 1 compute-minute + negligible disk]*

Purpose: This experiment validates that the feature names used in our ML models (claimed in **C5**) match the descriptions reported in Table 15 of our paper.

Preparation:
- Verify that the dataset files (e.g., `fbs_nas.csv`, `msa_nas.csv`) exist in the `dataset/` folder.

Execution:
- From the repository root, run:

```
python3 codes/feature-names.py
```

- The script writes the feature names to `outputs/column_names_output.txt` file.

Results:
- Compare each feature name to the entries in Table 15 of the paper

A.5 Notes on Reusability

Adapting to different protocols or attack types

- If you wish to collect new traces and annotate them with your target labels, our data loading and training pipelines can be reused with minimal changes, provided your CSV structure matches the expected format.
- To add new attacks, ensure that your labeling convention is clear (does not use the current labels (1-21)).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: SNI5GECT: A Practical Approach to Inject aNRchy into 5G NR

Shijie Luo, Matheus E. Garbelini, Sudipta Chattopadhyay, and Jianying Zhou
Singapore University of Technology and Design

A Artifact Appendix

A.1 Abstract

As part of our commitment to Open Science, we provide the full implementation of all components described in the SNI5GECT paper. This includes source code, binaries, and scripts necessary to evaluate 5G over-the-air sniffing and message injection using a standard PC (x86_64) running Ubuntu 22.04. Given the over-the-air nature of our evaluation, which relies on specific 5G target devices and a USRP Software Defined Radio (SDR), we also offer remote access to a machine equipped with this setup. Access is granted via SSH using a private key. In addition to the binaries, we provide the evaluation results and scripts required to reproduce the tables and figures presented in the SNI5GECT paper, by generating figures and terminal outputs. The evaluation procedure consists of a series of scripts, either included directly in the artifact or described in this appendix. Finally, the artifact features exploit scripts that demonstrate real-world attacks against Commercial Off-the-Shelf (COTS) 5G devices connected to the remote testbed.

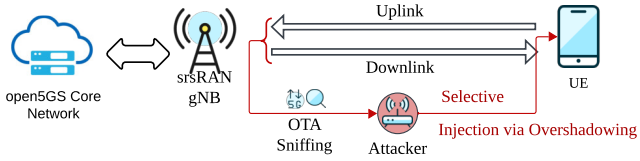


Figure 1: Overview of SNI5GECT evaluation

A.2 Description & Requirements

This artifact demonstrates the capabilities of our framework, including the setup for sniffing over-the-air 5G signals and injecting messages at specific stages of the communication process. To simplify testing, the remote evaluation machine is pre-configured with all necessary system libraries and dependencies required to run SNI5GECT. Hardware requirements are detailed in Section A.2.3, and instructions for building the system from source are provided in Section A.3.

A.2.1 Security, privacy, and ethical concerns

The test environment involves deploying a test 5G base station, which may cause nearby devices to connect to it. However, only devices using SIM cards configured with test Mobile Country Codes (MCC) and Mobile Network Codes (MNC) are affected. During evaluation, target phones may experience temporary effects such as modem crashes, fingerprinting, or network downgrade attacks. These effects are non-permanent and can be fully resolved by rebooting the device or reinserting the SIM card.

No persistent modifications are made to device firmware or operating systems, and no user data is accessed or stored at any point. We strongly recommend using dedicated test devices to avoid unintentional disruption to personal or production equipment.

A.2.2 How to Access

The released artifacts is publicly available on Zenodo: https://doi.org/10.5281/zenodo.15601773.
We also provide a server that has all the hardware and software dependencies. Access to the evaluation server is provided via a Cloudflare SSH tunnel. Please ensure that the cloudflared binary is installed on the client machine; it can be downloaded from the official Cloudflare website: https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/downloads/.

Next, configure SSH client by adding the following entry to ~/.ssh/config file:

```
Host sni5gect-artifacts
    HostName sni5gect.roskey.net
    User root
    ProxyCommand cloudflared access ssh --hostname %h
    IdentityFile ~/.ssh/artifact.key
```

Once configured, the reviewer can access the evaluation environment using:

```
chmod 0600 ~/.ssh/artifact.key
ssh sni5gect-artifacts
```

To transfer PDF or PCAP files to local machine, we recommend using SFTP. Additionally, the Docker container includes an x11vnc server for graphical access. Tools such as

RealVNC Viewer can be used to connect to the graphical interface at `localhost:5900` after setting up a port forwarding using ssh.

```
ssh -L 5900:localhost:5900 sni5gect-artifacts
```

A.2.3 Hardware dependencies

The following hardware components are required to evaluate the SNI5GECT framework:

- **USRP B210 SDR** – Used to run the legitimate srsRAN base station.
- **USRP B210 SDR** – Dedicated to the SNI5GECT framework for over-the-air sniffing and injection.
- **OnePlus Nord CE2** (Patch version: 2023-05-05) – Vulnerable User Equipment (UE) used for evaluation. (adb id: UWEUW4XG8XCA8PWS)
- **Pixel 7** (Patch version: 2023-05-05) – Vulnerable User Equipment (UE) used for evaluation. (adb id: 27211FDH20096Z)
- **Samsung S22** (Patch version: 2024-06-01) – Vulnerable User Equipment (UE) used for evaluation. (adb id: R5CT720QT7H)
- **Huawei P40** (Patch version: 2024-02-01) – Vulnerable User Equipment (UE) used for evaluation. (adb id: K5J0220312001992)
- **Fibocom FM150-AE USB modem** – Vulnerable User Equipment (UE) used for evaluation.

All listed devices are physically connected to the remote evaluation machine and pre-configured for remote experimentation.

A.2.4 Software dependencies

All software dependencies required to run the SNI5GECT framework are specified in the provided Dockerfile. These dependencies are intended to run inside a lightweight Ubuntu 22.04 Docker container without a GUI or other heavy background services. The main runtime components include:

- 5ghoul-5g-nr-attacks – commit `9739994` (included in the artifact).
- SNI5GECT– Modified from srsRAN 4G.
- srsRAN Project – Version `release_24_10_1` (used for the legitimate 5G base station).
- Open5GS – Core network implementation for srsRAN legitimate base station.
- Mongodb – Required by Open5GS for credential storage.
- QCSuper – Tools to communicate with the Qualcomm modems to get the raw frames received.

A.2.5 Benchmarks

We compared the DCI sniffing performance with NR-Scope srsran branch commit version: 2f30b0a3. https://github.com/PrincetonUniversity/NR-Scope

A.3 Getting Started

This section details the steps required to set up the same evaluation environment used in our experiments. To streamline the process and prevent dependency mismatches, we provide a pre-configured `Dockerfile` and `docker-compose.yml`. The `Dockerfile` initializes the environment using the `ubuntu:22.04` base image and installs all necessary components to run the SNI5GECT framework.

A.3.1 Installation (Optional)

To set up the environment, run the following commands from the root of the artifact directory. These steps will install all dependencies, compile the SNI5GECT framework, and start the srsRAN base station, Open5GS core network, and QCSuper in a fully functional state:

```
# Load a pre-build docker container
wget https://zenodo.org/records/15601773/files/sni5\
     gect-artifacts-docker.tar.gz
docker load < sni5gect-artifacts-docker.tar.gz
# Or build the docker container from scratch
wget https://zenodo.org/records/15601773/files/Sni5\
     Gect-source-code.zip
unzip Sni5Gect-source-code.zip
cd Sni5Gect-5GNR-sniffing-and-exploitation-main
docker compose build artifacts --build-arg GITHUB_TOKEN=\
     github_pat_11AH6MR7A02x31orqrUICh_uSzIpTz1u5syNaR2Q3\
     CThfLfofbjC94lKHfDRM9188QLFU2MHCNAViBTdG4
```

Next, download the evaluation results, update the path in the `docker-compose.yml` and start the docker container:

```
wget https://zenodo.org/records/15601773/files/sni5\
     gect-evaluation-results.zip
unzip sni5gect-evaluation-results.zip
docker compose up -d
```

Then to access the container, use the following:

```
docker exec -it artifacts bash
```

The folder structure inside the `/root` directory is listed as follows:

```
.
|-- sni5gect  # Sni5Gect project root folder
|   |-- bin -> /root/wdissector/bin
|   |-- build # Sni5Gect build output folder
|   |-- configs   # Configuration files for Sni5Gect
```

```
|   |-- logs      # Output for logs and PCAP files
|   |-- modules   # Sni5Gect exploit modules
|   |-- scripts   # Scripts to get sniffing performance
|   |-- shadower  # Source code for Sni5Gect framework
|   |   |-- hdr
|   |   |-- modules # Source code of exploit modules
|   |   |-- src
            # Broadcast Worker implementation
|   |   |   |-- broadcast_worker.cc
            # GNB DL Injector implementation
|   |   |   |-- gnb_dl_worker.cc
            # GNB UL Worker implementation
|   |   |   |-- gnb_ul_worker.cc
            # Distributes received subframes to components
|   |   |   |-- scheduler.cc
            # Syncher implementation
|   |   |   |-- syncer.cc
            # UE DL Worker implementation
|   |   |   |-- ue_dl_worker.cc
            # UE Tracker implementation
|   |   |   |-- ue_tracker.cc
            # wDissector wrapper
|   |   |   |-- wd_worker.cc
|-- srsran     # Evaluation legitimate 5G base station
|-- open5gs    # Evaluation legitimate Core network
|-- qcsuper    # QCSuper to intercept raw frames
|-- wdissector # wDissector dependency
# Evaluation results presented in the paper
|-- evaluation_results
|   |-- dci_evaluation # DCI sniffing result
|   |-- scripts        # Utils scripts
|   |-- sni5gect_5g_attacks  # Results for different \
     attacks and different phones
|   |-- sni5gect_injection_different_distance
|   |-- sni5gect_injection_different_state
|   |-- sni5gect_sniffing_srsran # Sniffing performance
|   |-- sni5gect_uplink_sniffing
|       |-- different_distance
|       |-- different_ta_offset
```

A.3.2 Basic Test of Radio Devices

To verify that the USRP B210 SDR devices are correctly connected to the host machine, run the following command:

```
uhd_find_devices
```

The expected output should list both SDR devices, similar to the following Figure 2. If the devices are not detected correctly, please ensure they are properly connected and powered. Next, update the configuration file at the following location:

```
/root/sni5gect/configs/config-srsran-n78-20MHz.conf
```

Ensure the USRP device is correctly specified, for example:

```
[source]
source_type = uhd
source_module = build/shadower/libuhd_source.so
source_params = type=b200,serial=3218CC4
```

Step 1: In terminal 1, start the Open5GS Core Network:

```
cd /root/open5gs
./build/tests/app/app -c open5gs.yaml
```

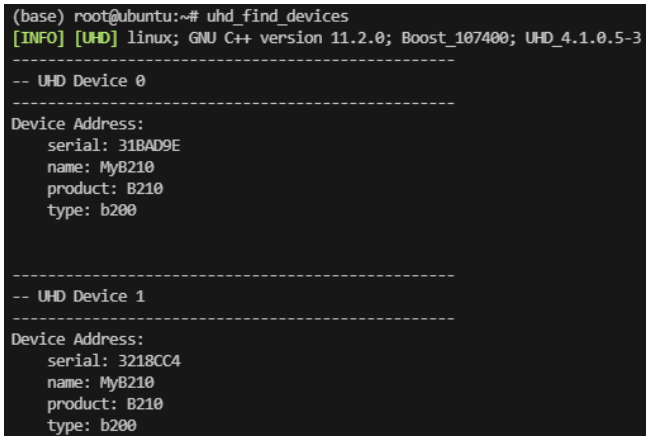


Figure 2: List UHD Devices

Step 2: Open a new terminal, start the srsRAN Base Station:

```
cd /root/srsran/
./build/apps/gnb/gnb -c srsran.conf
```

To confirm that the base station is running correctly, look for the line containing base station frequency information as shown in Figure 3.

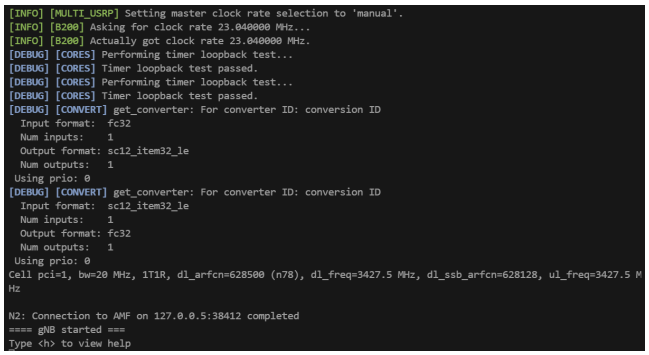


Figure 3: Successful started srsRAN

Step 3: Open a new terminal, start the SNI5GECT Framework as follows:

```
cd /root/sni5gect
./build/shadower/shadower \
configs/config-srsran-n78-20MHz.conf
```

If successful, the output in Figure 4, containing text `MIB applied to all workers` and `SIB1 applied to all workers` confirms that the SNI5GECT framework is running and ready to intercept phones connecting to the legitimate base station.

Step 4: To check if the phones are connected to the computer, open a new terminal and run the following command:

```
adb devices -l
```

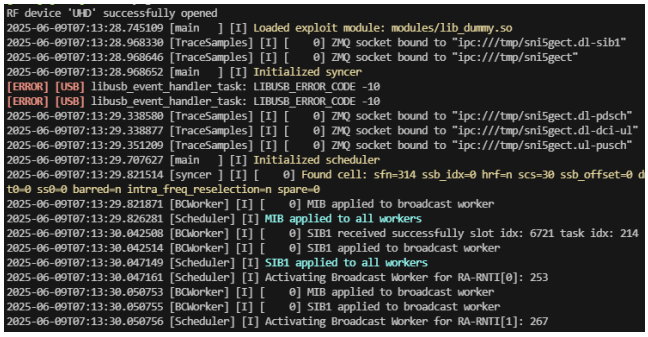


Figure 4: SNI5GECT pending UE connect

The following android devices in Figure 5 can be identified:

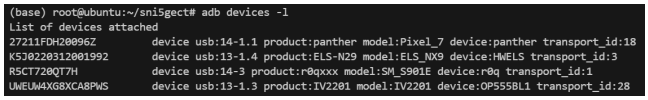


Figure 5: adb devices -l

If the device is connected correctly, the next step is to toggle the airplane mode and see if SNI5GECT captured the phone connecting to the base station.

To turn off airplane mode (connecting to 5G base station):

```
adb -s UWEUW4XG8XCA8PWS shell cmd connectivity \
    airplane-mode disable
```

To check the 5G connection status, please use the following command:

```
adb -s UWEUW4XG8XCA8PWS shell dumpsys telephony.registry |\
     grep mTelephonyDisplayInfo=TelephonyDisplayInfo

# Example Output: Connected to 5G NR
mTelephonyDisplayInfo=TelephonyDisplayInfo {network=NR, \
    overrideNetwork=NONE, isRoaming=false}

# Example Output: Connected to LTE/Emergency Service
mTelephonyDisplayInfo=TelephonyDisplayInfo {network=LTE, \
    overrideNetwork=NONE, isRoaming=false}

# Example Output: No connection
mTelephonyDisplayInfo=TelephonyDisplayInfo {network=\
    UNKNOWN, overrideNetwork=NONE, isRoaming=false}
```

To turn on airplane mode (disconnecting from 5G base station), use the following:

```
adb -s UWEUW4XG8XCA8PWS shell cmd connectivity \
    airplane-mode enable
```

On the SNI5GECT terminal in example Figure 6, if we can observe that SNI5GECT have detected a new UE with the keyword `Found new UE with tc-rnti`, then we can confirm that each component works well.

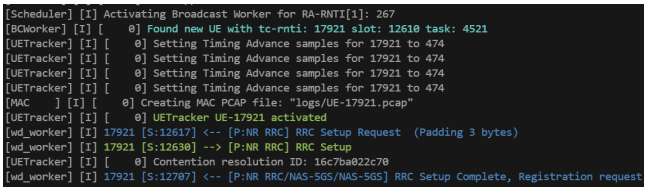


Figure 6: SNI5GECT detected UE

A.4 Evaluation workflow

In this section, we outline the experimental workflow used to evaluate the SNI5GECT framework. The evaluation is structured around three core objectives, each targeting a specific capability of the system. We describe how to reproduce these evaluations using the provided setup and explain how to generate the same figures and tables presented in the paper.

1. **DCI Sniffing Performance**: We begin by evaluating the SNI5GECT framework's ability to detect Downlink Control Information (DCI). DCIs are essential for scheduling both uplink and downlink transmissions, and their reliable decoding is a prerequisite for further message interpretation. This step assesses the robustness and accuracy of DCI detection.

2. **Message Sniffing Performance**: Building on the DCI results, we then assess the overall sniffing capabilities of the SNI5GECT framework. This includes decoding actual downlink and uplink messages exchanged between the User Equipment (UE) and the legitimate base station. This evaluation reveals how effectively the framework can reconstruct ongoing 5G communications and track session states.

3. **Injection and Attack Performance**: Finally, we evaluate the framework's active capabilities. Using QCSuper, we confirm whether the injected messages are successfully received by the USB modem. In addition to confirmation via QCSuper, we also perform over-the-air injection attacks in real-time. This phase demonstrates the effectiveness of SNI5GECT in manipulating live 5G communication between the base station and vulnerable UEs.

A.4.1 Major Claims

In this section, we summarize the major claims presented in the paper and highlight the corresponding expected evaluation outcomes.

(C1): Reliable DCI Sniffing Performance — We claim that SNI5GECT can reliably sniff Downlink Control Information (DCI), which forms the foundation for subsequent decoding of downlink and uplink messages. We expect the DCI sniffing success rate to exceed 90%, as shown in Figure 10.

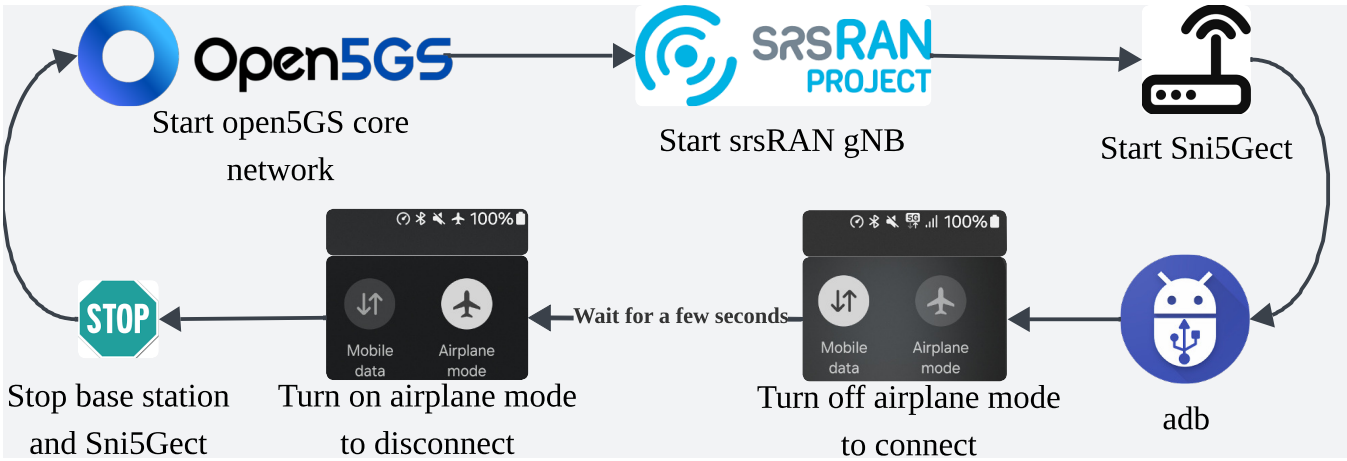


Figure 7: SNI5GECT Evaluation steps

(C2): Robust Sniffing Performance — We claim that the SNI5GECT framework offers robust overall sniffing performance. The expected overall message sniffing accuracy is around 80%, with uplink message decoding often achieving around 70% accuracy. These results correspond to Table 1, Figure 13, and Figure 14.

(C3): Effective Injection Capabilities — We claim that SNI5GECT can effectively inject messages into the communication stream of a target UE. The expected injection success rate can achieve around 60% or higher, as demonstrated in Table 3 and Table 2. This enables a range of successful attacks leveraging the injection capability. Results are shown in Figure 26 and Figure 27.

A.4.2 Evaluation Steps

The overall evaluation steps are shown in the Figure 7. In this section, we describe in detail about how to run each step.

1. **Clear the logs folder:** The logs folder should contain only the logs from the current evaluation session. Residual logs from previous sessions may affect the accuracy of the evaluation results. Please clear the logs folder before starting any experiment using the following command:

```
rm -rf /root/sni5gect/logs/*
```

2. **Start open5GS core network:** The open5GS core network is required to run the legitimate 5G base station. Firstly open a terminal and please refer to the following command to start the open5GS.

```
cd /root/open5gs
./build/tests/app/app -c open5gs.yaml
```

If the open5GS core network fails to start and shows that port 7777 is in use, please use the following command to kill existing open5gs and run the command above to start the open5GS core network again:

```
sudo pkill -9 app
sudo pkill -9 open5gs
```

The logs will be stored in the following file `/root/sni5gect/logs/open5gs.log`

3. **Start srsRAN gNB:** After the open5GS core network starts correctly, open a new terminal and use the following command to start the srsRAN base station.

```
cd /root/srsran/
./build/apps/gnb/gnb -c srsran.conf
```

The result `gnb.log` and `mac_nr.pcap` will be stored in folder `/root/sni5gect/logs/` for later evaluation.

4. **Start SNI5GECT:** After the srsRAN base station starts correctly, open a new terminal and start the SNI5GECT framework and pipe the command line output to the logs folder for later evaluation.

```
cd /root/sni5gect
./build/shadower/shadower configs/config-srsran-n78-20MHz.conf | tee logs/sni5gect.log
```

5. **Toggle airplane mode:** The airplane mode can be easily controlled via adb command. Please note that there are multiple phones connected to the server, so the evaluators have to specify the adb id to control a specific phone.

```
# Disable airplane mode (connect to 5G base station)
adb -s UWEUW4XG8XCA8PWS shell cmd connectivity \
    airplane-mode disable

# Turn on airplane mode (disconnect from 5G base \
    station)
adb -s UWEUW4XG8XCA8PWS shell cmd connectivity \
    airplane-mode enable
```

6. **Stop the base station and SNI5GECT:** Simply press Ctrl+C to stop each component.

A.4.3 Experiments

This section describes the experiments conducted to evaluate and support each claim. We also demonstrate how to use the provided scripts to analyze the provided test data and reproduce the tables and figures presented in the paper.

(E1): DCI Sniffing Evaluation: [60 human-minutes]

This experiment evaluates the DCI sniffing performance of the SNI5GECT framework by comparing the number of correctly sniffed DCI messages with the ground truth obtained from the srsRAN legitimate base station.

How to: Connect a UE (e.g., a phone) to the legitimate base station. Once SNI5GECT detects the target UE, it begins sniffing the DCI messages exchanged between the base station and the UE. After collecting data for 5 to 10 seconds, turn on the airplane mode, stop both SNI5GECT and the base station. (This is because some evaluation devices may unexpectedly drop their connection after approximately 15 seconds. The recommended 5–10 second capture window helps ensure that the connection remains stable throughout the evaluation period.) Then, compare the number of correctly sniffed DCIs against the ground truth logs generated by the base station.

Preparation: Edit the configuration file:

```
/root/sni5gect/configs/config-srsran-n78-20MHz.conf
```

Set the logging level for the worker to DEBUG and ensure the dummy exploit module is used for passive sniffing:

```
[log]
log_level = INFO
syncer_log_level = INFO
worker_log_level = DEBUG # Change the worker log \
    level to debug to retrieve the DCI information
bc_worker_log_level = INFO

[exploit]
module = modules/lib_dummy.so
```

Execution: The command used to run the evaluation is provided in Section A.4.2. Please following the command provided to run the evaluation and collect the logs. After data collection, stop both the base station and the SNI5GECT framework.

An example command-line output is shown in Figure 8. From the white text, we can observe the transmitted DCI information, including their formats, frequency and time allocations, as well as the modulation and coding scheme. The subsequent line demonstrates how SNI5GECT utilizes this scheduling information from the DCI to attempt decoding the actual messages.

Results: The logs for the above evaluation will be stored in the folder `/root/sni5gect/logs`. Please use the following script to compute the DCI sniffing success rate, it will give out the total number of DCIs from srsRAN, number of DCIs sniffed by SNI5GECT and the corresponding success rate in the terminal output. Example output is shown in Figure 9.

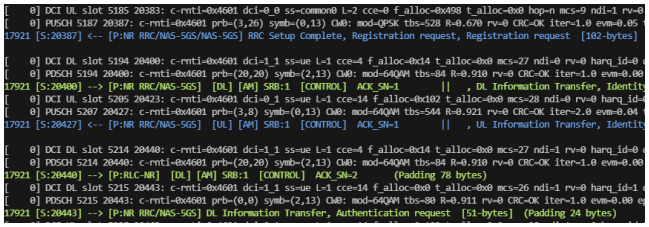


Figure 8: DCI sniffing example output

```
cd /root/sni5gect
python3 scripts/get_dci_sniffing_performance.py
```

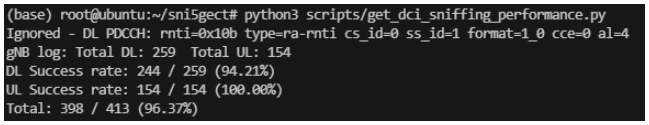


Figure 9: Get DCI sniffing performance example output

Figure 10 corresponds to **Figure 6: Success Rate of DCI Search** in the paper. Data and scripts are located at `/root/evaluation_results/dci_evaluation/`

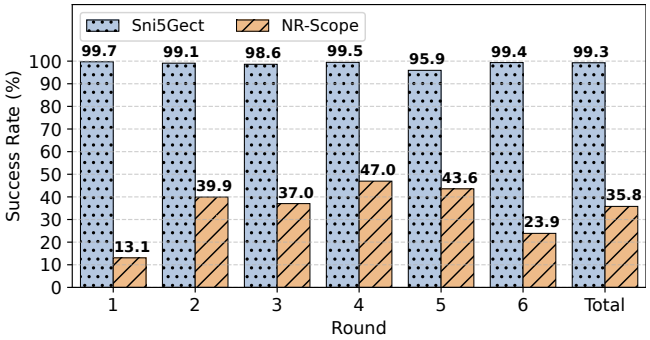


Figure 10: Success Rate of DCI search

To evaluate the existing result and plot the performance as shown in the paper, run the following commands:

```
cd /root/evaluation_results

# Get the DCI sniffing performance for Sni5Gect
python3 dci_evaluation/compare_sni5gect.py
# Result stored in: /root/evaluation_results/\
    dci_evaluation/sni5gect_eval_result.csv

# Get the DCI sniffing performance for NR-Scope
python3 dci_evaluation/compare_nrscope.py
# Result stored in: /root/evaluation_results/\
    dci_evaluation/nrscope_eval_result.csv
```

```
# Plot the figure based on the CSV data
cd /root/evaluation_results/dci_evaluation
python3 plot_result.py
# Result plot: /root/evaluation_results/\
    dci_evaluation/dci_sniffing_success_rate.pdf
```

(E2): Message Sniffing Evaluation: [60 human-minutes] This experiment evaluates the message sniffing performance of the SNI5GECT framework. It compares the decoded downlink and uplink messages with the ground truth collected from the srsRAN base station to measure overall accuracy.

How to: Connect a UE (phone) to the legitimate base station. Once SNI5GECT detects the target UE, it begins sniffing both downlink and uplink messages from the over-the-air signals exchanged between the base station and the UE. After collecting data for 5 to 10 seconds, stop both SNI5GECT and the base station. The resulting PCAP file will be stored in the `/root/sn5gect/logs/` directory. Finally, compare the sniffed messages with the ground truth obtained from the base station.

Preparation: Configure the `worker_log_level` to `INFO` and ensure the dummy exploit module is used for passive sniffing:

```
[log]
log_level = INFO
syncer_log_level = INFO
worker_log_level = INFO
bc_worker_log_level = INFO

[exploit]
module = modules/lib_dummy.so
```

Execution: To evaluate the performance of message sniffing, the process is the same as what we have shown in DCI sniffing. Please refer to the same commands presented in Section A.4.2. An example command-line output is shown in Figure 11. In the output, downlink messages sent from the base station to the UE are highlighted in green, while uplink messages sent from the UE to the base station are highlighted in blue.

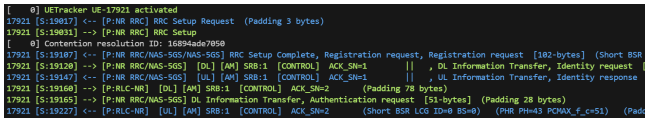


Figure 11: Sniffing Example output

Results: The logs for the above evaluation will be stored in the folder `/root/sni5gect/logs`, with the messages sniffed by SNI5GECT stored in the file with name for example `UE-17921.pcap`. Please use the following script to compute the message sniffing success rate, it will give out the total number of messages sent between srsRAN and the UE, as well as the messages sniffed by SNI5GECT and the corresponding success rate in the terminal output. Example output is shown in Figure 12.

```
cd /root/Sni5Gect
python3 scripts/get_sniffing_performance.py
```

```
(base) root@ubuntu:~/sni5gect# python3 scripts/get_sniffing_performance.py
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Parsing Sni5Gect pcap file: /root/sni5gect/logs/UE-17921.pcap
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
RNTI: 17921 MSIN: 0000064953:
DL Success rate: 222 / 222 (100.00%)
UL Success rate: 118 / 132 (89.39%)
```

Figure 12: Get sniffing performance example output

The paper presents three categories of message sniffing performance evaluation:

1. **Table 1**: Corresponds to **Table 2** in the paper and demonstrates the effectiveness of SNI5GECT in sniffing both downlink and uplink messages.
2. **Figure 13**: Corresponds to **Figure 4** in the paper and shows the success rate of uplink sniffing with respect to the distance between the UE and the base station.
3. **Figure 14**: Corresponds to **Figure 5** in the paper and illustrates the impact of varying TA (Timing Advance) command offsets on uplink sniffing performance.

Table 1: Effectiveness of SNI5GECT sniffing capability for different distance between the UE and the SNI5GECT SDR. Number of sniffed messages is provided alongside accuracy.

Device	Dist.	Sniffing Accuracy	Uplink	Downlink
OnePlus Nord CE 2	0 m	93.32% (4418)	75.77% (938)	99.54% (3480)
	1 m	92.20% (8074)	70.45% (1576)	99.66% (6498)
Huawei P40 Pro	0 m	94.86% (1180)	86.08% (371)	99.51% (809)
	1 m	94.74% (1262)	84.87% (387)	99.89% (875)
Pixel 7	0 m	86.55% (1094)	75.89% (491)	97.73% (603)
	1 m	90.71% (1768)	74.11% (435)	97.87% (1333)
Samsung Galaxy S22	0 m	93.15% (1061)	88.78% (467)	96.90% (594)
	1 m	97.51% (745)	94.68% (338)	100% (407)

Table 1: This table summarizes the effectiveness of SNI5GECT in capturing both downlink and uplink messages. The evaluation was conducted with the UE positioned at two distances from the base station: 0 meters and 1 meter. The corresponding experimental results are stored in the following directory:

```
/root/evaluation_results/sni5gect_sniffing_srsran
```

To regenerate the CSV file from the raw log data, run the following script, which parses the experiment results and generates the message sniffing performance metrics:

```
cd /root/evaluation_results
python3 sni5gect_sniffing_srsran/\
    get_sniffing_performance.py
# Output file: /root/evaluation_results/sni5\
    gect_sniffing_srsran/sniffing_performance.csv
```

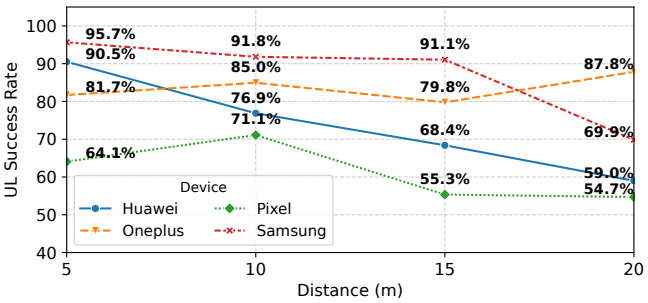


Figure 13: Success Rate of Uplink Sniffing w.r.t distance

Figure 13: Success Rate of Uplink Sniffing w.r.t distance This figure illustrates the success rate of uplink message sniffing as the distance between the UE and the sniffer increases. The raw experimental data used to generate this figure is stored in the following directory:

```
/root/evaluation_results/sni5gect_uplink_sniffing/\
    different_distance
```

To compute the success rate and generate Figure 13, execute the following scripts:

```
cd /root/evaluation_results/sni5gect_uplink_sniffing
python3 parse_logs_different_distance.py
# Output different_distance_detail.csv

python3 plot_different_distance.py
# Result figure: uplink_sniffing_different_distance.\
    pdf
```

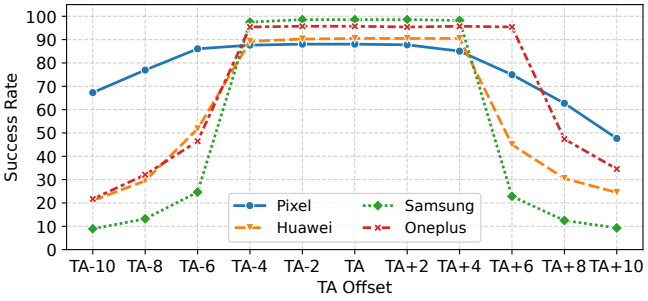


Figure 14: Uplink Sniffing w.r.t TA command offset

Figure 14: Uplink Sniffing w.r.t TA command offset This figure presents the uplink message sniffing success rate under varying Timing Advance (TA) command offsets. The raw data used for this analysis is located in the following directory:

```
/root/evaluation_results/sni5gect_uplink_sniffing/\
    different_ta_offset
```

To process the logs, calculate the success rates, and generate Figure 14, execute the following commands:

```
cd /root/evaluation_results/sni5gect_uplink_sniffing
python3 parse_logs_different_ta_offset.py
# Output different_ta_offset_detail.csv

python3 plot_different_ta.py
# Result figure: uplink_sniffing_different_ta.pdf
```

(E3): Message Injection Evaluation: [120 human-minutes] This experiment measures the effectiveness of the SNI5GECT framework in injecting messages towards a victim device. Using QCSuper and a USB modem, we confirm successful message reception by the victim UE.
How to: In this evaluation, we utilize QCSuper, a tool capable of interfacing directly with Qualcomm-based USB modems to capture raw over-the-air frames. This enables us to verify whether specific messages are received by the target UE. We use `cutecom` to interact with the USB modem over the serial interface in order to manually initiate and terminate 5G connections.
Preparation: To perform message injection during the 5G attach procedure, different exploit modules are required depending on the message type and protocol state. For example, to inject a NAS message immediately after a `Registration Request`, the module for `Registration Reject` should be used, since the initial connection typically begins with the UE sending a `Registration Request` message.

```
[exploit]
module = modules/lib_dg_registration_reject.so
```

Execution: First, start the base station as described in Section A.4.2. Then, launch QCSuper with the following command. This will capture a packet trace in `qcsuper.pcap` and display a live view of incoming frames from the USB modem. The reviewer can use client such as VNCViewer to connect to the vnc server and observe the live traffic.

```
cd /root/qcsuper
python3 qcsuper.py --usb-modem auto --pcap-dump /root\
    /sni5gect/logs/qcsuper.pcap --wireshark-live
```

Next, open a separate terminal and start `cutecom` to communicate with the USB modem via its serial interface. This can be done in the GUI from the vnc session which allows sending AT commands to trigger connection and disconnection events on the target UE.
Alternatively, you can use the terminal-based application `minicom`. To start a new session with `minicom`, run the following command:

```
minicom -D /dev/ttyUSB2
# Exit minicom:
# Ctrl+A followed by X, then press Enter to exit \
    minicom
```

Please refer to the following command to toggle the airplane mode of the USB modem.

```
# Check the modem starts and connects correctly
AT  # Expecting: OK
```

```
# Run cell search
AT+cops=?

# Connect to 5G
AT+cops=1,2,00101,12

# Disconnect from base station
AT+cops=2
```

An example command-line output is shown in Figure 15. After receiving the `Registration Request` message, the UE releases the connection. Following a timeout period, the base station sends an `RRC Release` message to terminate the connection.

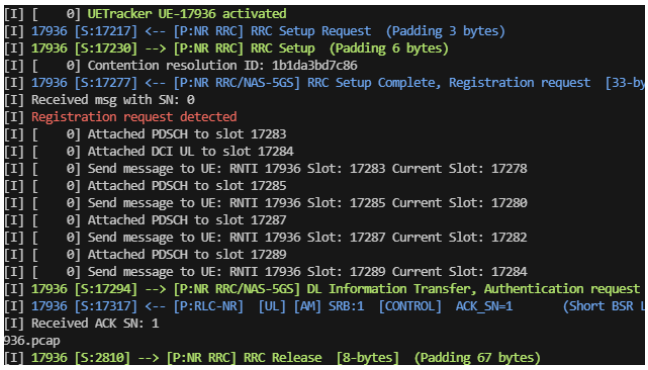


Figure 15: Message Injection stdout Example

An example result captured from QCSuper is shown in Figure 16. From the screenshot, we can confirm that the UE receives and accepts our injected `Registration Reject` message with the cause `N1 Mode not Allowed`.



Figure 16: Message Injection QCSuper Example

Results: The evaluation process above will store the result `qcsuper.pcap` to folder `/root/sni5gect/logs` by default. To get the success rate of the injection performance, please simply use the following command. This will print the total number of UE accepted injections over the total number of injections, as well as the success rate in percentage on the standard output terminal. The example output is shown in Figure 17

```
cd /root/sni5gect
python3 scripts/get_injection_performance.py
```

There are two sets of injection evaluation results presented in the paper:

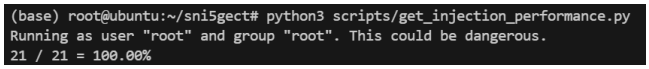


Figure 17: Get message injection performance

- **Table 3** corresponds to **Table 3: SNI5GECT Injection Performance**.
- **Table 2** corresponds to **Table 4: Injection Performance at Different Distances**.

Table 3 evaluates the injection success rate and message duplication behavior at various states of the 5G connection procedure. The raw experiment data are stored in the following directory:

```
/root/evaluation_results/sni5\
    gect_injection_different_state
```

To regenerate the CSV file used in the paper and produce the summarized table, run the following command:

```
# Result for column RRC Setup Request
cd /root/evaluation_results/sni5\
    gect_injection_different_state/\
    injection_after_rrc_setup
python3 evaluate.py

# Result for column Registration Request
cd /root/evaluation_results/sni5\
    gect_injection_different_state/\
    inject_after_registration_request
python3 evaluate.py

# Result for column Authentication Failure
cd /root/evaluation_results/sni5\
    gect_injection_different_state/\
    inject_after_authentication_failure
python3 evaluate.py

# Result for column Security Mode Complete
cd /root/evaluation_results/sni5\
    gect_injection_different_state/\
    inject_after_security_mode_complete
python3 evaluate.py
```

Table 2: Injection Performance at Different Distances.

Distance	Total	Success	Success Rate
5m	73	70	95.89%
10m	65	61	93.85%
15m	73	69	94.52%
20m	73	61	83.56%

Table 2 evaluates how physical distance between the attacker and the target UE impacts the injection success rate. The raw data are stored in the directory:

```
/root/evaluation_results/sni5\
    gect_injection_different_distance
```

To regenerate the CSV file containing the success and total counts for each round at different distances, run:

```
cd /root/evaluation_results/sni5\
    gect_injection_different_distance
python3 parse_logs.py
# Result: result.csv
```

(E4): Example Attacks Evaluation: [Several hours] This experiment demonstrates real-world attacks that leverage the message injection capabilities of the SNI5GECT framework. These attacks are conducted against a live victim UE to validate the framework's end-to-end exploitation potential.

How to: In this evaluation, we trigger the phones to connect to the target base station. At specific states during the connection process, SNI5GECT injects carefully crafted or intercepted messages into the communication stream. This can induce various effects, such as protocol downgrades, device fingerprinting, modem crashes, or authentication bypasses.

Preparation: For each type of attack, use a dedicated attack module. Refer to the list below for the appropriate modules corresponding to each attack. Uncomment the relevant line to enable the desired exploit module. Please refer to Table 4 for the function of each module and please note that only one exploit module should be loaded at a time.

```
# Sniffing
module = modules/lib_dummy.so
# Fingerprint
module = modules/lib_identity_request.so
# Sniff and extract Authentication Request
module = modules/\
    lib_dg_authentication_request_sniffer.so
# Downgrade Attacks
module = modules/lib_dg_authentication_replay.so
module = modules/lib_dg_registration_reject.so
# 5Ghoul attacks: crash the OnePlus MTK modem
module = modules/lib_mac_sch_mtk_rlc_crash.so
module = modules/lib_mac_sch_mtk_rrc_setup_crash_3.so
module = modules/lib_mac_sch_mtk_rrc_setup_crash_4.so
module = modules/lib_mac_sch_mtk_rrc_setup_crash_6.so
module = modules/lib_mac_sch_mtk_rrc_setup_crash_7.so
# Authentication Bypass: affect Pixel 7 Exynos modem
module = modules/lib_plaintext_registration_accept.so
```

Execution: Update the exploit modules as needed, then start the base station and SNI5GECT as described in Section A.4.2. Toggle airplane mode on the device to trigger a new connection.

```
# Trigger the phone to connect to the base station
adb -s UWEUW4XG8XCA8PWS shell cmd connectivity \
    airplane-mode disable
# Wait for a few seconds
# Disconnect the phone from the base station
adb -s UWEUW4XG8XCA8PWS shell cmd connectivity \
    airplane-mode enable
```

Results: The behaviors and impacts of different attacks vary. Some attacks result in effects such as fingerprinting, downgrade, or authentication bypass.

Registration Reject Attack: After the User Equipment (UE) sends a `Registration Request` message, SNI5GECT injects a `Registration Reject` message. Upon receiving and accepting this message, the UE disconnects from the base station. Subsequently, the base station may repeatedly attempt to resend messages such as `Identity Request`, `Authentication Request`, or `Security Mode Command`, but receive no response from the UE. An example command-line output is shown in Figure 18. We can see SNI5GECT sniffed the injected message `Registration Reject`, we can also observe the base station kept sending `Authentication Request` after the UE disconnects. If subsequent messages such as `Authentication Response` or `Security Mode Complete` appear, or if there are excessive occurrences of `UL-SCH` or `DL-SCH`, it indicates that the attack has failed. Please toggle airplane mode and attempt the attack again.

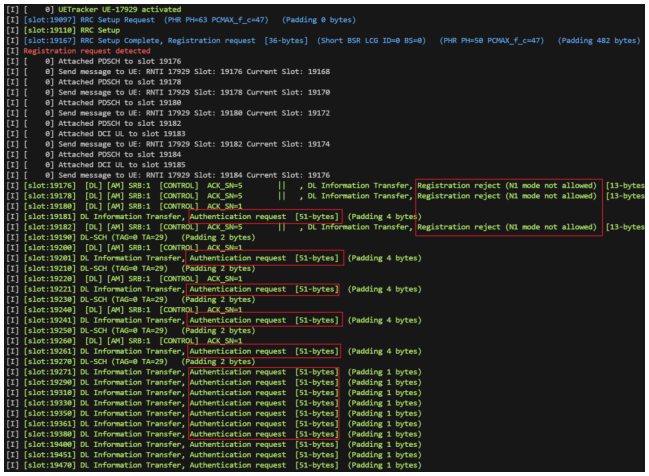


Figure 18: Registration Reject Example output

To run the evaluation, please follow the steps in Section A.4.2. The result is expected to be stored in `/root/sni5gect/logs` folder. To get the success rate of such attack, please use the following command. This will give out the success rate of `Registration Reject` attack in standard output terminal similar to Figure 19.

```
python3 scripts/get_registration_reject_performance.\
    py
```

Identity Request Attack: Following the UE's transmission of a `Registration Request` message, SNI5GECT injects an `Identity Request` message. When the victim UE receives and processes this message, it responds with an `Identity Response`. SNI5GECT then sniffs the `Identity Response` and extracts the `SUCI`, demonstrating a fingerprinting attack. An example command-line output is provided in Figure 20, we can observe the base station sends `Security Mode Command`, but the

Table 3: SNI5GECT Injection Performance.

Duplication #	RRC Setup Request			Registration Request			Authentication Failure			Security Mode Complete		
	# Messages	# Injected	Success Rate	# Messages	# Injected	Success Rate	# Messages	# Injected	Success Rate	# Messages	# Injected	Success Rate
1	356	306	85.96%	232	181	78.02%	47	29	61.70%	51	23	45.10%
2	251	219	87.25%	423	337	79.67%	72	60	83.33%	67	60	89.55%
3	316	272	86.08%	350	262	74.86%	42	28	66.67%	46	44	95.65%
4	234	202	86.32%	364	297	81.59%	71	51	71.83%	43	39	90.70%

Module File	**Description / Notes**
`lib_dummy.so`	Pure sniffing; no injection occurs.
`lib_identity_request.so`	Fingerprinting attack: Injects Identity Request and expects an Identity Response.
`lib_dg_authentication_request_sniffer.so`	Sniffs the Authentication Request message.
`lib_dg_authentication_replay.so`	Authentication replay attack: After sniffing, update the relevant file `shadower/modules/dg_authentication_replay.cc` with the captured Authentication Request and recompile (`ninja -C build`). **Note:** After a successful attack, the UE may not reconnect to the base station for 5+ minutes. Reboot the phone using `adb -s {id} reboot`.
`lib_dg_registration_reject.so`	Downgrade attack: Sends Registration Reject to trigger immediate downgrade.

`lib_mac_sch_mtk_rlc_crash.so`
`lib_mac_sch_mtk_rrc_setup_crash_3.so`
`lib_mac_sch_mtk_rrc_setup_crash_4.so`
`lib_mac_sch_mtk_rrc_setup_crash_6.so`
`lib_mac_sch_mtk_rrc_setup_crash_7.so`
	5Ghoul attacks: Crash attacks affecting OnePlus devices with MTK modems.

`lib_plaintext_registration_accept.so`	Authentication bypass: Only affects Pixel 7 devices with Exynos modem.

Table 4: Attack Modules and Their Descriptions

```
(base) root@ubuntu:~/sni5gect# python3 scripts/get_registration_reject_performance.py
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
Running as user "root" and group "root". This could be dangerous.
13 / 13 = 100.00%
```

Figure 19: Get Registration Reject Performance Output

```
[    0] UETracker UE-17922 activated
[slot:10777] RRC Setup Request  (PHR PH=63 PCMAX_f_c=47)   (Padding 0 bytes)
[slot:10790] RRC Setup
[    0] Contention resolution ID: 1271a2896270
[slot:10867] RRC Setup Complete, Registration request, Registration request  [94-bytes]  (Short BSR LCG ID=0
Sent identity request
[    0] Attached PDSCH to slot 10876
[    0] Send message to UE: RNTI 17922 Slot: 10876 Current Slot: 10868
[    0] Attached PDSCH to slot 10878
[    0] Send message to UE: RNTI 17922 Slot: 10878 Current Slot: 10870
[    0] Attached PDSCH to slot 10880
[    0] Send message to UE: RNTI 17922 Slot: 10880 Current Slot: 10872
[slot:10876]  [DL] [AM] SRB:1  [CONTROL]  ACK_SN=1        ||   , DL Information Transfer, Identity request
[slot:10878]  [DL] [AM] SRB:1  [CONTROL]  ACK_SN=1        ||   , DL Information Transfer, Identity request
[slot:10880]  [DL] [AM] SRB:1  [CONTROL]  ACK_SN=1
[slot:10885] Security Mode Command  [9-bytes]  (Padding 3 bytes)
[slot:10907]  [UL] [AM] SRB:1  [CONTROL]  ACK_SN=1        ||   , UL Information Transfer, Identity response
Identity: 0000064952
[slot:10920]  [DL] [AM] SRB:1  [CONTROL]  ACK_SN=2       (Padding 78 bytes)
[slot:10950] DL-SCH (TAG=0 TA=29)   (Padding 81 bytes)
[slot:12325] RRC Release  [8-bytes]  (Padding 71 bytes)
```

Figure 20: Identity Request Example output

```
(base) root@ubuntu:~/sni5gect# python3 scripts/get_identity_request_performance.py
6 / 10 = 60.00%
```

Figure 21: Get Identity Request Performance Output

UE replies with `Identity Response` and the extracted SUCI is shown in red. If this red message does not appear, it indicates that either the injection has failed or the sniffer did not successfully capture the response. Please toggle airplane mode to establish a new connection and rerun the attack.

To run the evaluation, please follow the steps in Section A.4.2. The result is expected to be stored in `/root/sni5gect/logs` folder. To get the success rate of such attack, please use the following command. This will give out the success rate of `Identity Request` attack in standard output terminal similar to Figure 21.

```
python3 scripts/get_identity_request_performance.py
```

Authentication Replay Attack: This is the most complex exploit in our evaluation and involves two stages: sniffing and replaying. In the first stage (sniffing), the module `lib_dg_authentication_request_sniffer.so` is used to capture a legitimate `Authentication Request` message sent from the base station to the UE. This sniffing step only needs to be performed once per specific UE; the captured `Authentication Request` can be reused in subsequent replay attempts.

In the second stage (replaying), when a `Registration Request` message is received, SNI5GECT replays the previously captured `Authentication Request`. The UE responds with an `Authentication Failure` (cause: `Synch Failure`) and starts timer T3520. After sniffing this response, SNI5GECT updates the RLC and PDCP sequence numbers and replays the `Authentication Request` again. This process may repeat multiple times. Eventually, after timer T3520 expires, the UE deems that the network has failed the authentication check, it locally releases the connection, and bars the active cell. If no other 5G base station is available, the UE ignores the currently available cell, downgrades to 4G and remains in downgraded state for 300 seconds, as specified in 3GPP TS 24.501 version 16.5.1 Release 16, section 5.4.1.2.4.5 (Abnormal cases in the UE). In the example output shown in Figure 22, the UE can be seen replying with the `Authentication Failure` message twice, and after replaying the `Authentication Request` message the third time, the UE disconnects, and the base station is keeps sending `RRC Release`. If messages such as `Authentication Response` or `UE Capability Enquiry`—which indicate progression in the connection—appear, or if there are excessive occurrences of `UL-SCH` or `DL-SCH`, it means the attack has failed. Please toggle airplane mode and try the attack again.

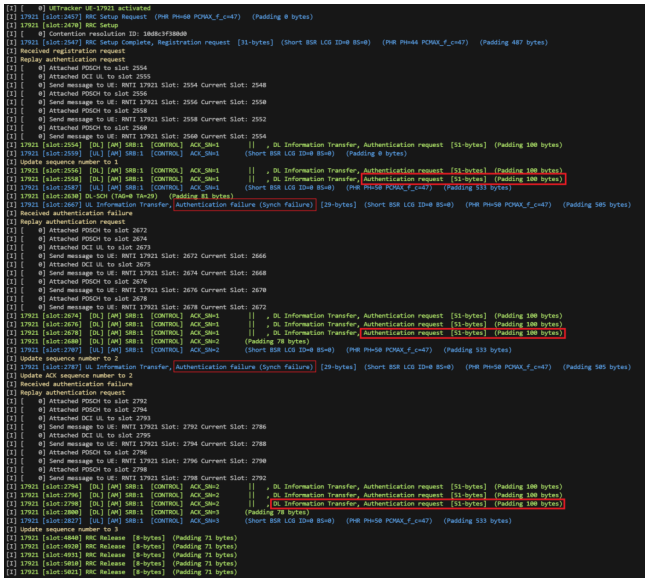


Figure 22: Authentication Replay Example output

To run the evaluation, please follow the steps in Section A.4.2. Please note that in this evaluation, we suggest using the open5GS from the wdissector directory.

```
cd /root/wdissector/
./3rd-party/open5gs-core/build/tests/app/app -c \
    configs/5gnr_gnb/open5gs.yaml
```

The result is expected to be stored in `/root/sni5gect/logs` folder. To get the success rate of such attack, please use the following command, which considers the attack as successful when `Authentication Failure` is received more than one time and the connection is not successfully established. This will give out the success rate of `Authentication Request` replay attack in standard output terminal.

```
python3 scripts/get_auth_replay_performance.py
```

5Ghoul Attacks: These attacks demonstrate how vulnerabilities discovered by the 5Ghoul framework can be exploited to crash target UE modems using SNI5GECT, resulting in Denial-of-Service (DoS) conditions.

```
# CVE-2023-20702
# module = modules/lib_mac_sch_mtk_rlc_crash.so
# CVE-2023-32843
# module = modules/lib_mac_sch_mtk_rrc_setup_crash_3.\
    so
# CVE-2023-32842
# module = modules/lib_mac_sch_mtk_rrc_setup_crash_4.\
    so
# CVE-2024-20003
# module = modules/lib_mac_sch_mtk_rrc_setup_crash_6.\
    so
# CVE-2023-32845
# module = modules/lib_mac_sch_mtk_rrc_setup_crash_7.\
    so
```

The current implementation targets OnePlus phones specifically. To prepare the device for testing, you may toggle airplane mode using the following command:

```
# Disable airplane mode to trigger connection
adb -s UWEUW4XG8XCA8PWS shell cmd connectivity \
    airplane-mode disable
# Enable airplane mode to drop connection
adb -s UWEUW4XG8XCA8PWS shell cmd connectivity \
    airplane-mode enable
```

The core idea of these attacks is to transmit malformed RLC or RRC Setup messages to the target UE. Upon processing these malformed messages, the UE crashes immediately. Full details of these exploits are available on the official 5Ghoul disclosure page: https://asset-group.github.io/disclosures/5ghoul/ To verify whether an attack has succeeded, monitor the device logs using logcat. Specifically, look for crash-related keywords such as `sModemReason`. Open a new terminal and run the following command to begin monitoring:

```
# Clear existing logcat logs
adb -s UWEUW4XG8XCA8PWS logcat -c
# Monitor logcat logs
adb -s UWEUW4XG8XCA8PWS logcat -b radio,crash,system,\
    main | grep sModemReason
```

```
# module = modules/lib_mac_sch_mtk_rlc_crash.so
# 07-04 10:51:37.365  1622  2198 D \
    MDMKernelUeventObserver: sModemReason:fid:457077\
    782;cause:[Fatal error(MPU_NOT_ALLOW)] err_code1\
    :0x0000001D err_code2:0x910D66F6 err_code3:0x910\
    D66E2 CaDeFa Supported

# module = modules/lib_mac_sch_mtk_rrc_setup_crash_3.\
    so
# 07-04 11:04:51.807  1622  2198 D \
    MDMKernelUeventObserver: sModemReason:fid:156734\
    6682;cause:[ASSERT] file:mcu/l1/nl1/internal/md9\
    7/src/rfd/nr_rfd_configdatabase.c line:4380 p1:0\
    x00000001

# module = modules/lib_mac_sch_mtk_rrc_setup_crash_4.\
    so
# 07-04 11:06:50.880  1622  2198 D \
    MDMKernelUeventObserver: sModemReason:fid:372552\
    4241;cause:[ASSERT] file:mcu/l1/mml1/mml1_endc/\
    src/mml1_endc_db_hdlr.c line:524 p1:0x91920c70

# module = modules/lib_mac_sch_mtk_rrc_setup_crash_6.\
    so
# 07-04 11:09:46.846  1622  2198 D \
    MDMKernelUeventObserver: sModemReason:fid:105190\
    3049;cause:[ASSERT] file:mcu/l1/nl1/internal/md9\
    7/src/ctrl/nr_ctrl_mgm.c line:16460 p1:0x0000000\
    0

# module = modules/lib_mac_sch_mtk_rrc_setup_crash_7.\
    so
# 07-04 11:10:42.422  1622  2198 D \
    MDMKernelUeventObserver: sModemReason:fid:100627\
    7484;cause:(MSONIC0) [ASSERT] file:dsp3/\
    coresonic/msonic/modem/brp/nr/nr_brp/src/\
    nr_brp_top_irq.c line:927
```

If the attack is successful, you will observe crash indicators, including the `sModemReason` keyword, as shown in Figure 23 and Figure 24. A large number of red error messages may appear, and the SNI5GECT tool itself may crash—this is expected behavior. Since SNI5GECT also receives the malformed RRC Setup message it generates, it may attempt to decode and apply the invalid configuration.

Please note: after a successful attack, the UE's modem will crash and may not reconnect to the base station immediately. Recovery typically requires either a waiting period (e.g., one minute) or manually rebooting the device to restore 5G connectivity.

5G AKA Bypass Attack: This attack is implemented in the `lib_plaintext_registration_accept.so` attack module, located in the modules folder. It is currently applicable only to the Pixel phone in our testbed. Please use the following command to toggle airplane mode on the Pixel device:

```
# Disable airplane mode to trigger connection
adb -s 27211FDH20096Z shell cmd connectivity \
    airplane-mode disable
# Enable airplane mode to drop connection
adb -s 27211FDH20096Z shell cmd connectivity \
    airplane-mode enable
```

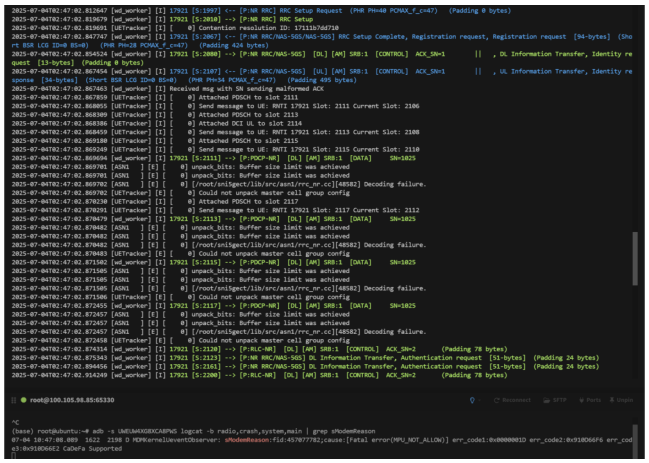


Figure 23: MTK RLC Crash

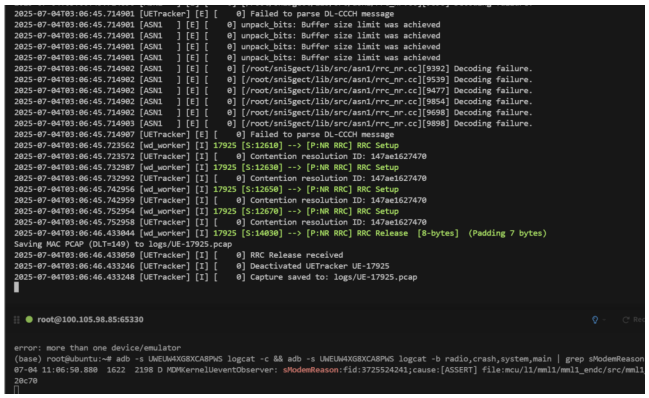


Figure 24: MTK RRC Setup Crash

The core logic of the attack is as follows: after the UE transmits a `Registration Request` message, SNI5GECT injects a `Registration Accept` message. Upon accepting this message, the victim UE responds with `Registration Complete` and a `PDU Session Establishment Request`. Since the core network receives these unexpected messages, it instructs the gNB to release the connection by sending an `RRC Release` message, thereby immediately terminating the connection. We can observe the aforementioned process in Figure 25. If messages such as `Authentication Response` or `Security Mode Complete` are observed and the UE proceeds with the connection, it indicates that the attack has failed. Please toggle airplane mode to establish a new connection and rerun the attack.

Evaluation results presented in the paper: The evaluation result presented in the paper is stored in the following structure in:

```
/root/evaluation_results/sni5gect_5g_attacks
```

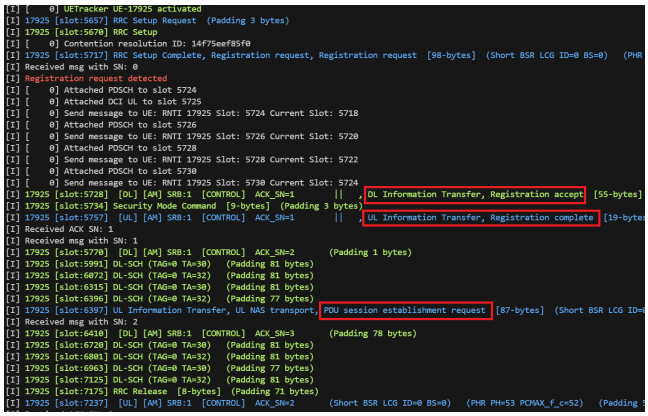


Figure 25: 5G AKA Bypass Example output

```
# Organized as <phone>_<base_station>/<exploit>_<\
    distance>m
.
|-- 5Ghoul_srsran
|   |-- mtk_rlc_crash
|   |-- mtk_rrc_setup_crash_3
|   |-- mtk_rrc_setup_crash_4
|   |-- mtk_rrc_setup_crash_6
|   |-- mtk_rrc_setup_crash_7
|-- 5Ghoul_effnet
|-- oneplus_srsran
|   |-- authentication_replay_0m
|   |-- authentication_replay_1m
|   |-- identity_request_0m
|   |-- identity_request_1m
|   |-- registration_reject_0m
|   |-- registration_reject_1m
|-- oneplus_effnet
|   |-- authentication_replay_0m
|   |-- authentication_replay_1m
|   |-- identity_request_0m
|   |-- identity_request_1m
|   |-- registration_reject_0m
|   |-- registration_reject_1m
|-- huawei_srsran
|-- huawei_effnet
|-- pixel_srsran
|-- pixel_effnet
|-- samsung_srsran
|-- samsung_effnet
|-- registration_accept
```

To generate the same table as presented in the paper, please refer to the following command:

```
cd /root/evaluation_results
python3 sni5gect_5g_attacks/\
    calculate_registration_reject_success_rate.py
# Result CSV: identity_request_results.csv

python3 sni5gect_5g_attacks/\
    calculate_registration_reject_success_rate.py
# Result CSV: registration_reject_results.csv

python3 sni5gect_5g_attacks/\
    calculate_authentication_replay_success_rate.py
# Result CSV: authentication_request_results.csv
```

Figure 26 corresponds to **Figure 7: Attack Success Rates by Device, Distance, and Attack Type using srsRAN as legitimate gNB** and Figure 27 corresponds to **Figure 8: Attack Success Rates by Device, Distance, and Attack Type using Effnet as legitimate gNB**. To plot these two figures, please refer to the following commands:

```
cd /root/evaluation_results/sni5gect_5g_attacks
python3 update_attack_json_data.py
# Format the data for plot

python3 plot_srsran_result.py
# data_attack_success_rate_srsran.pdf

python3 plot_effnet_result.py
# data_attack_success_rate_effnet.pdf
```

A.5 Notes on Reusability

SNI5GECT provides a tool for DCI sniffing, message sniffing and message injection attacks. Additionally, this can be useful in the following tasks:

1. **Attack evaluation without rogue base station:** Unlike approaches that relies on a rogue base station that is assumed to have man-in-the-middle capabilities, the SNI5GECT framework enables evaluation of 5G attacks by passively sniffing and stealthily injecting messages into live sessions. This allows the researchers to evaluate their attacks under more realistic conditions.

2. **Throughput analysis:** This can be done by utilizing the steps shown in DCI sniffing evaluation. In particular, such an evaluation can provide the DCIs sent from the base station to the UE, from which we can learn the amount of traffics communicated between the base station and the UE. This makes it useful for throughput analysis over-the-air.

3. **Traffic analysis:** This can be accomplished by utilizing the message sniffing capability shown in message sniffing evaluation. For example, the user can analyze the traffic between the base station and the UE, to research on privacy tracking related topics.

4. **Anomaly detection:** This artifact provides a significant amount of PCAP from both the sniffer and the base station during the message injection attacks. To the best of our knowledge, this is the first open artifact providing such dataset with over-the-air injection attacks. This can be potentially used for anomaly detection tasks.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodol-

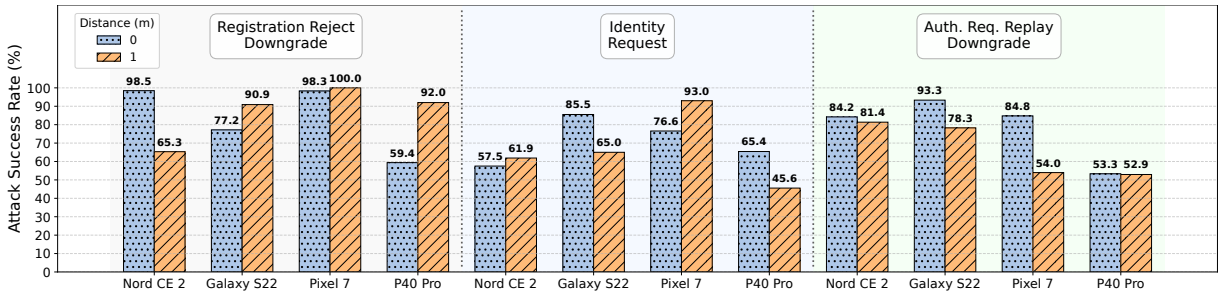


Figure 26: Attack Success Rates by Device, Distance, and Attack Type using **srsRAN** as legitimate gNB.

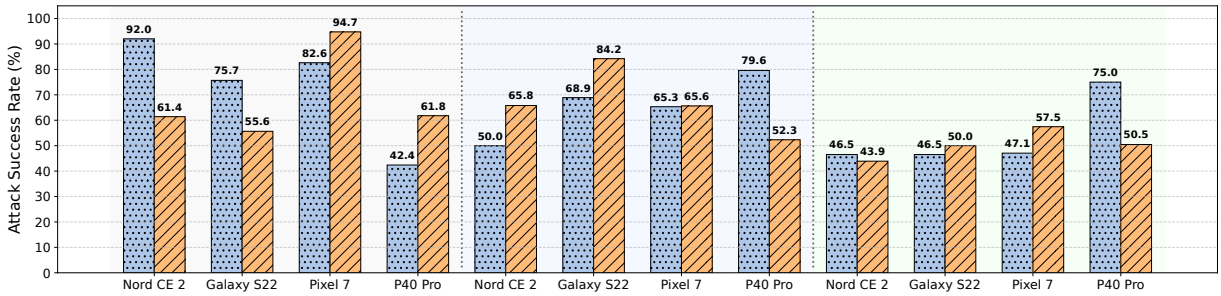


Figure 27: Attack Success Rates by Device, Distance, and Attack Type using **Effnet** as legitimate gNB.

ogy followed for the evaluation of this artifact can be found at
https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Disparate Privacy Vulnerability: Targeted Attribute Inference Attacks and Defenses

Ehsanul Kabir
Pennsylvania State University

Lucas Craig
Pennsylvania State University

Shagufta Mehnaz
Pennsylvania State University

A Artifact Appendix

A.1 Abstract

As machine learning (ML) technologies become increasingly prevalent in privacy-sensitive domains such as healthcare and finance, they raise critical concerns about potential privacy leakage risks. This paper investigates one such threat: attribute inference attacks, where adversaries exploit public, non-sensitive attributes to infer private, sensitive information. While these attacks may perform poorly on an entire dataset, they can achieve high accuracy for records belonging to specific vulnerable groups, a phenomenon termed disparate vulnerability. This work introduces a novel disparity inference attack to identify high-risk groups and two targeted variations of attribute inference attacks that significantly outperform untargeted versions. Additionally, we propose an innovative defense mechanism, the Balanced Correlation Defense (BCorr), which effectively mitigates these risks while preserving model utility. For artifact evaluation, our goals are to ensure reproducibility of the experiments, validate the implementation of the proposed attacks and defenses, and enable further exploration of disparate vulnerability in ML systems. The provided artifacts include datasets, a modular codebase, and detailed Jupyter notebooks for reproducing key results, such as the relationship between correlation and attack performance, the efficacy of targeted attacks, and the mitigation impact of BCorr. These artifacts aim to support open science practices, ensuring accessibility, transparency, and reusability for the research community.

A.2 Description & Requirements

This section outlines the experimental setup, including security considerations, access instructions, software dependencies, hardware requirements, and benchmark datasets for reproducibility.

A.2.1 Security, privacy, and ethical concerns

There are no known risks to machine security, data privacy, or ethical concerns associated with executing this artifact. The provided code does not involve destructive steps, nor does it disable any security mechanisms.

A.2.2 How to access

The artifact is available at https://zenodo.org/records/14732956. Please refer to 'Version v2' of the artifact for the most up-to-date version.

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

The artifact requires a Linux, macOS, or Windows operating system with Python 3.9.16 or later. A virtual environment can be set up using Conda or Python's `venv`. All necessary dependencies are listed in `requirements.txt` and can be installed using `pip install -r requirements.txt`. Running Jupyter notebooks requires installing `notebook` via `pip install notebook`. No proprietary software is required, and all dependencies are open-source and can be installed using standard package managers.

A.2.5 Benchmarks

The experiments in this artifact rely on the following datasets: `census19.csv`, which contains the Census-19 dataset, and `Adult_35222.csv` and `Adult_10000.csv`, which contain the Adult dataset partitions for training and testing, respectively. Additionally, the Texas-100X dataset is required. To ensure reproducibility, we have attached a preprocessed version of the dataset with the filename `texas_100_cleaned.csv`.

A.3 Set-up

A.3.1 Installation

To install and set up the artifact, first create a virtual environment using Conda or Python's `venv` and activate it. Then, install all required dependencies using `pip install -r requirements.txt`. To run Jupyter notebooks, install `notebook` via `pip install notebook`. Create a directory `<PATH_TO_MODELS>` to store models after training.

A.3.2 Basic Test

To verify the setup, open and run all cells in the Jupyter notebook `basic_test.ipynb`. This notebook loads and preprocesses the datasets for four different experiment scenarios. A successful run will produce the correlation values for different groups in each scenario in the final cell.

A.4 Evaluation workflow

This section outlines the steps required to evaluate the artifact, validate its functionality, and reproduce the key results presented in the paper. It consists of two parts: Major Claims, which enumerate the core findings supported by the experiments, and Experiments, which describe the operational steps needed to test these claims. Each experiment is linked to a corresponding claim and includes details on execution, estimated compute time, and expected outcomes.

A.4.1 Major Claims

(C1): The correlation between the sensitive attribute and the output is a significant factor in the vulnerability to attribute inference attacks. Specifically, datasets with high correlation are more vulnerable than datasets with low correlation. This is proven by the experiment (E1) described in section 4.1 of the main paper whose results are illustrated in Figure 1.

(C2): Correlation influences disparate vulnerability among groups i.e. groups with high correlation are more vulnerable to attribute inference attacks than groups with low correlation. Additionally, the correlation level influences the confidence score distribution and can be estimated using angular difference. This is proven by the experiment (E2) described in section 4.2 of the main paper whose results are illustrated in Figure 2 and Figure 3.

(C3): Imputation attack performance degrades if the auxiliary dataset size is small or the marginal prior of the auxiliary dataset is different from the target data. Additionally, practical imputation attack (ImpP) performance is poorer than CSMIA/LOMIA in high correlation groups. This is proven by the experiment (E3) described in section 6.2 of the main paper whose results are illustrated in Figure 4.

(C4): Disparity inference attack can rank groups in terms of vulnerability more efficiently than the baseline attack using auxiliary dataset. This is proven by the experiment (E4) described in section 6.3 of the main paper whose results are reported in Table 1.

(C5): Single Attribute-based and Nested Attribute-based targeted attribute inference attacks can achieve higher attack success rate than their untargeted counterpart. Specifically, the attack performance gradually increases with lower attack budget (κ). This is proven by the experiment (E5) described in section 6.4 of the main paper whose results are reported in Table 2 and Table 3.

(C6): Our proposed defense BCorr can effectively mitigate disparate vulnerability without reducing model utility or introducing fairness concerns. Specifically, ASRD is negligible after applying BCorr and there is a minimal change in model performance and fairness metrics such as Equalized Odds Difference (EOD) and Demographic Parity Difference (DPD) do not increase. This is proven by the experiment (E6) described in section 7.2 of the main paper whose results are reported in Table 5.

A.4.2 Experiments

(E1): [3 compute-hour]: This experiment evaluates the impact of correlation between sensitive attributes and model outputs on the success of various attribute inference attacks. Using the Census19 and Texas-100X datasets, we generate 19 training sets per dataset, each with varying levels of correlation. We then train target models and perform CSMIA, LOMIA, Imputation, and Neuron Importance attacks. The results are expected to indicate that CSMIA and LOMIA exhibit a strong positive relationship with correlation, achieving higher attack success rates as correlation increases.
Preparation: No additional preparation required.
Execution: Run all cells in the Jupyter notebook `correlation_vs_attack_performance.ipynb`
Results: The last two cells of the notebook is expected to generate plots similar to Figure 1.

(E2): [2 compute-hour]: This experiment examines the impact of correlation on disparate vulnerability by analyzing how attack performance varies across different groups within the Census-19 dataset. Specifically, we vary correlation levels in Male and Female groups across nine scenarios, observing that groups with higher correlation experience greater attack success rates. Additionally, we introduce the concept of angular difference by analyzing confidence score distributions using hexagonal binning. The result is expected to reveal that attack vulnerability is closely tied to correlation, with distinct trajectory patterns in confidence scores across different groups with varying correlation.
Preparation: No additional preparation required.
Execution: Run all cells in the Jupyter notebook `angular_difference_by_sex.ipynb`
Results: The last two subsections of the notebook are expected to generate plots similar to Figure 2 and Figure 3 respectively. Specifically, in each scenario, the group with the higher correlation should show higher attack success rate than the group with the lower correlation. Additionally, angular difference is plotted for groups with varying correlation (-0.4, -0.5, -0.6) and a trend of increase in angular difference should be observed with higher correlation value.

(E3): [1 compute-hour]: The experiment evaluates the impact of distributional drift in auxiliary datasets on the performance of imputation attacks. Two types of drift are analyzed: dataset-level distributional drift, where the marginal prior and dataset size is varied to assess how deviations from the original training data affect attack performance, and group-level distributional drift, where correlations within occupation groups differ from the overall dataset correlation. The result is expected to demonstrate that imputation attacks perform well only when the auxiliary dataset closely matches the original data distribution. In cases where marginal prior deviates significantly or group-level correlations shift, imputation attacks perform worse than CSMIA and LOMIA, particularly in highly vulnerable groups.
Preparation: No additional preparation required.
Execution: Run all cells in the Jupyter notebook `imputation_vs_ai_aux_size_and_distrib_diff.ipynb`
Results: The subsection 'Plot Results' and the last subsection of the notebook are expected to generate plots similar to Figure 4(a) and Figure 4(b) respectively. Specifically, the generated table should show lower attack performance as dataset size decreases or marginal prior decreases. The plot should report lower attack performance for ImpP than CSMIA/LOMIA for groups 'Adm-clerical', 'Prof-specialty', and 'Tech-support'.

(E4): [1 compute-hour]: This experiment evaluates the effectiveness of the proposed disparity inference attack, which ranks groups based on their vulnerability to attribute inference attacks. To assess the ranking quality, the experiment uses Kendall's Tau and Spearman's Rank Correlation, comparing the disparity inference attack against a baseline ranking method that relies on an auxiliary dataset.
Preparation: No additional preparation required.
Execution: Run all cells in the Jupyter notebook `disparity_inference.ipynb`
Results: The last two cells of the notebook are expected to generate results similar to Table 1. Specifically, the ranking quality (Spearman's R/Kendall's Tau) of disparity inference attack should be significantly better (closer to ± 1) than baseline.

(E5): [1 compute-hour]: This experiment evaluates two proposed variations of targeted attribute inference attacks—single attribute-based and nested attribute-based—using the Census19, Texas-100X, and Adult datasets. Groups are selected based on specific attributes, with the nested approach refining selection through multiple attributes. The result is expected to show that targeted attacks consistently outperform untargeted approaches, with refined selection improving inference accuracy.
Preparation: No additional preparation required.
Execution: Run all cells in the Jupyter notebook `targeted_attribute_inference.ipynb`
Results: The subsections 'Single Attribute-based Targeted Attacks' and 'Nested Attribute-based Targeted Attacks' of the notebook are expected to generate results similar to Table 2 and Table 3 respectively. Specifically, there should be a trend in attack performance increase with lower values of κ and higher depth. Note that the untargeted attack performance in Census19 and Texas-100X may be slightly different from the ones reported in the paper. This may be caused by the randomization involved in the sampling process for these dataset specific scenarios. However, the claim in our paper that targeted attacks achieve increasingly better performance with smaller target sets is expected to uphold.

(E6): [5 compute-hour]: This experiment evaluates the effectiveness of BCorr in mitigating disparate vulnerability across different groups using the Census-19 and Texas-100X datasets. The defense is tested on both binary attributes (`SEX` and `SEX_CODE` respectively) and multi-valued attributes (`ST` and `PAT_STATUS` respectively). For comparison, a Fairness Constraint-based defense (FC) is also applied as a baseline. The evaluation measures attack success rate disparity (ASRD), group fairness (EOD, DPD), and model accuracy (MA) to assess the impact of BCorr on both security and fairness. The result is expected to show that BCorr significantly reduces disparities across groups while maintaining model utility and preserving fairness.
Preparation: No additional preparation required.
Execution: Run all cells in the Jupyter notebook `balancing_corr_defense.ipynb`
Results: The last cell of the notebook is expected to generate results similar to Table 5. ASRD with BCorr should be significantly smaller than without BCorr across all scenarios. MA with BCorr is expected to be similar to MA without BCorr. EOD and DPD with BCorr should not be higher than EOD and DPD without BCorr.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Enhanced Label-Only Membership Inference Attacks with Fewer Queries

Hao Li*
Institute of Software,
Chinese Academy of Sciences

Zheng Li*
Shandong University

Siyuan Wu
Institute of Software,
Chinese Academy of Sciences

Yutong Ye
Institute of Software,
Chinese Academy of Sciences

Min Zhang†
Institute of Software,
Chinese Academy of Sciences

Dengguo Feng
Institute of Software,
Chinese Academy of Sciences

Yang Zhang
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

In this artifact, we demonstrate a label-only Membership Inference Attack called DHAttack, designed for Higher performance and Higher stealth, focusing on the boundary distance of individual samples to mitigate the effects of sample diversity, and measuring this distance toward a fixed point to minimize query overhead. The code primarily comprises three files: preprocessData.py, trainTargetModel.py and DHAttackBase.py. These files support the CIFAR10 and CIFAR100 datasets, as well as the VGG-16, ResNet-56, and MobileNetV2 models.

In this appendix, we summarize the key claims of the paper and demonstrate how they can be verified using this artifact. Our first claim is that DHAttack achieves effective results with fewer than 100 queries. Secondly, DHAttack achieves superior performance in most cases. It is important to note that completing all experiments takes several hundred hours. This is because our attack process requires training 256 reference models. While attacking a single target model only requires training these 256 reference models once (a process that takes tens of hours), verifying all six target models in the paper (excluding those in the appendix) requires six times that duration. In addition, Algorithm 2 in our paper defines the fixed sample as an input parameter, allowing it to be modified. Rather than directly using 'RGB-255', we assign the maximum pixel value across all dimensions to the sample (which is also an outlier) as a fixed point in this artifact, primarily for implementation convenience.

*The first two authors made equal contributions.
†Corresponding author.

A.2 Description & Requirements

This section outlines the experimental setup, including hardware and software requirements and relevant benchmarks used to produce results.

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns associated with executing this artifact. All experiments are conducted solely on public datasets and widely used model architectures, ensuring that no sensitive or private data is involved.

A.2.2 How to access

Our artifact is at https://zenodo.org/records/14728863.

A.2.3 Hardware dependencies

Evaluating our artifact requires an NVIDIA GeForce RTX 2080 Ti or a more powerful GPU. Since GPU performance directly affects runtime, we recommend using an NVIDIA GeForce RTX 4090 for optimal efficiency. There are no strict requirements for the CPU or memory, though we recommend at least 64GB of RAM to ensure smooth execution.

A.2.4 Software dependencies

Our artifact is compatible with Windows. All necessary dependencies are listed in the requirements.txt file, with detailed installation instructions available in Section A.3.1.

A.2.5 Benchmarks

Our experiment requires the CIFAR10 and CIFAR100 datasets, which must be downloaded from their official web-

sites and placed in the designated folder. Detailed configuration instructions are provided in Section A.3.1.

A.3 Set-up

This section outlines the installation and configuration steps required to set up the environment for artifact evaluation.

A.3.1 Installation

The installation process is primarily divided into three stages: dependency installation, data preparation, and target model training.

Dependency installation. Create a new Python environment using Conda, specifying the environment name and Python version:

```
conda create -n env_name python=3.8.
```

Once the environment is created, activate it and use pip to install all the packages listed in the "`requirements.txt`" file. To do this, first activate the environment with:

```
conda activate env_name,
```

then run the following command to install the dependencies:

```
pip install -r requirements.txt.
```

Alternatively, you can manually install the dependencies listed in the "`requirements.txt`" file.

Data preparation. For CIFAR10, please place the CIFAR10 dataset files (downloaded from the official website, including "`data_batch_1`" to "`data_batch_5`" and "`test_batch`" files) into the "`.\data\cifar-10-download`" folder. For CIFAR100, please place the CIFAR100 dataset files (downloaded from the official website, including "`test`" and "`train`" files) into the "`.\data\cifar-100-download`" folder. In addition, for Linux systems, we provide two helper scripts, "`download_cifar10.sh`" and "`download_cifar100.sh`", which can automatically download the CIFAR10 and CIFAR100 datasets from the official website and place them into the appropriate directories.

Next, execute the following commands:

```
python preprocessData.py --dataset CIFAR10.
python preprocessData.py --dataset CIFAR100.
```

Target model training. Please run the commands to obtain the target models VGG-16, ResNet-56 and MobileNetV2 trained on CIFAR10:

```
python trainTargetModel.py --dataset CIFAR10
--classifierType vgg --num_epoch 100
python trainTargetModel.py --dataset CIFAR10
--classifierType resnet --num_epoch 100
python trainTargetModel.py --dataset CIFAR10
--classifierType mobilenet --num_epoch 100
```

Then, by changing the "`dataset`" parameter and running the three commands again, you can obtain target models trained on the CIFAR100 dataset. This process is expected to take approximately one hour.

A.3.2 Basic Test

You can use the code "DHAttackBase.py" we provide to verify that all required dependencies are installed correctly.

```
python DHAttackBase.py --dataset CIFAR10
--classifierType mobilenet
--num_epoch_for_refmodel 100 --disturb_num 5
```

If the process shows that reference models are being trained, it indicates that all dependencies have been installed correctly. At this point, you can safely terminate the process, as full execution would take tens of hours. Note that this command is only used to verify whether the required dependencies are correctly installed. Its performance will be low due to the "`disturb_num`" being set to only 5. For the formal experiment, please refer to Section A.4.

A.4 Evaluation workflow

In this section, we present the main claims of the paper along with the experiments that support them.

A.4.1 Major Claims

The major claims presented in our paper are as follows:

(C1): DHAttack achieves effective results with fewer than 100 queries. This is proven by the experiment (E1) described in Section 5.1, whose results are illustrated in Figures 5 and 6.

(C2): DHAttack achieves superior performance in most cases. This is proven by the experiment (E2) described in Section 5.1, whose results are illustrated in Table 3.

A.4.2 Experiments

The experiments supporting the main claims are detailed as follows:

(E1): [High Stealth] [30 human-minutes + 450(75*6) compute-hour + 220GB disk]: This experiment aims to verify that DHAttack can achieve an effective attack with only a small number of queries to the target model. We expect performance to fluctuate at query counts of 5, 10, 20, 30, 50, 100, or 200, but DHAttack generally achieves optimal results with fewer than 100 queries. As the query count increases (e.g., 100 or 200), the fixedBD measurement in our method becomes more sensitive to decision boundary complexity, causing unstable values across different reference models and affecting attack performance. Therefore, Figures 5 and 6 highlight performance fluctuations due to varying query numbers and demonstrate that optimal effectiveness is achieved with a small number of queries.

How to: To run the experiment, execute DHAttackBase.py, which generates one AUC value and one TPR at 0.001 FPR per run. Use the `classifierType` and `dataset` parameters to specify the target model and dataset (e.g., vgg and CIFAR10 correspond to the first subfigure in the top-left corner of Figures 5 and 6). After successfully attacking a specified target model and dataset for the first time, 256 reference models will have been trained (a process taking tens of hours). For subsequent attacks on the same target model and dataset, this step can be skipped by setting the trainRefModel parameter.

Preparation: If you have successfully completed data preparation and target model training in A.3.1, no further setup is required.

Execution: Please run the command:

```
python DHAttackBase.py --dataset CIFAR10
--classifierType vgg --num_epoch_for_refmodel
 100 --disturb_num 5
```

This process is expected to take approximately 60 hours. Once completed, record the AUC and the TPR at 0.001 FPR, as reported by our artifact. Next, run the following commands (a process that may take approximately ten hours or longer) and record the results for each:

```
python DHAttackBase.py --dataset CIFAR10
--classifierType vgg --num_epoch_for_refmodel
 100 --disturb_num 10 --trainRefModel False

python DHAttackBase.py --dataset CIFAR10
--classifierType vgg --num_epoch_for_refmodel
 100 --disturb_num 20 --trainRefModel False

python DHAttackBase.py --dataset CIFAR10
--classifierType vgg --num_epoch_for_refmodel
 100 --disturb_num 30 --trainRefModel False

python DHAttackBase.py --dataset CIFAR10
--classifierType vgg --num_epoch_for_refmodel
 100 --disturb_num 50 --trainRefModel False

python DHAttackBase.py --dataset CIFAR10
--classifierType vgg --num_epoch_for_refmodel
 100 --disturb_num 100 --trainRefModel False

python DHAttackBase.py --dataset CIFAR10
--classifierType vgg --num_epoch_for_refmodel
 100 --disturb_num 200 --trainRefModel False
```

These results will contribute to the DHAttack curve in the first subfigure of Figures 5 and 6, where the target model is VGG-16 trained on CIFAR10. Repeat these commands while adjusting the `classifierType` (i.e., resnet and mobilenet) and `dataset` (i.e., CIFAR100) parameters to obtain other performance curves for DHAttack.

Results: The `disturb_num` parameter specifies the number of queries made to the target model. You can then observe that when attacking each target model, DHAttack achieves optimal attack effectiveness with fewer than 100 queries. Moreover, in most cases, the performance is expected to surpass the baseline results reported in Figures 5 and 6.

(E2): [High Performance] [20 human-minutes + 20 compute-hour + 220GB disk]: This experiment aims to verify that DHAttack can achieve superior performance in most cases. Our expected results are generally consistent with those shown in Table 3. There may be slight fluctuations in the results. These differences are primarily due to the inherent randomness in training the reference models, such as random initialization. However, these variations do not impact the advantages of our approach.

How to: To run the experiment, execute DHAttackBase.py, which generates one AUC value and one TPR at 0.001 FPR per run. Set the dataset parameter to CIFAR10 and vary the classifierType between vgg, resnet, and mobilenet to obtain the results presented in Table 3.

Preparation: If you have successfully completed E1, no further setup is required.

Execution: Please run the commands:

```
python DHAttackBase.py --dataset CIFAR10
--classifierType vgg --num_epoch_for_refmodel
 100 --disturb_num 30 --trainRefModel False

python DHAttackBase.py --dataset CIFAR10
--classifierType resnet
--num_epoch_for_refmodel 100 --disturb_num
 50 --trainRefModel False

python DHAttackBase.py --dataset CIFAR10
--classifierType mobilenet
--num_epoch_for_refmodel 100
--disturb_num 50 --trainRefModel False
```

After executing each command, record the AUC and the TPR at 0.001 FPR, as reported by our artifact.

Results: The `dataset` and `classifierType` parameters define the target model, while the `disturb_num` parameter specifies the number of queries made to the target model. You can then observe that when attacking each target model, DHAttack achieves superior performance in most cases, consistent with the results presented in Table 3.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Preventing Automated Monitoring on Voice Data

Irtaza Shahid, Nirupam Roy

University of Maryland, College Park
{irtaza, niruroy}@umd.edu

A Artifact Appendix

A.1 Abstract

We present the artifact for the paper, titled "For Human Ears Only: Preventing Automated Monitoring on Voice Data". It includes the datasets, code, and models necessary for training and reproducing the major results presented in the paper. As voice communication becomes an essential part of modern life, the exposure of sensitive information through audio calls presents significant privacy risks. Malicious actors can gain access to this data by compromising user devices, exploiting communication channels, or attacking data servers, making it vulnerable to automated monitoring systems that can identify speakers and extract speech content. To address these privacy concerns, we introduce VoiceSecure, the first microphone module designed to prevent automated monitoring of speech while preserving its natural sound for humans. By leveraging the principles of human auditory perception, VoiceSecure employs a set of speech modifications that are adaptively tuned in real-time to obscure speaker identity and speech content, without compromising the audio quality for humans.

This artifact supports the goals of open science and reproducibility by providing all necessary components to replicate the core results of the paper. It includes source code, pre-trained models, processed datasets, evaluation metrics (WER, MMR, STOI), and scripts to generate all key figures and tables. The artifact allows evaluators to apply VoiceSecure transformations, measure their impact on speech recognition and speaker identification systems, and verify that the transformations preserve intelligibility—thereby validating the paper's major claims.

A.2 Description & Requirements

This section lists all the information necessary to recreate the same experimental setup we have used to develop our system.

A.2.1 Security, privacy, and ethical concerns

Our artifact presents no known security, privacy, or ethical risks to evaluators or their systems. It does not require administrative access, disabling of any security mechanisms, or the use of potentially harmful scripts or binaries.

A.2.2 How to access

The complete artifact is permanently archived and accessible via Zenodo at the following DOI: https://doi.org/10.5281/zenodo.15603263 This archive includes:

- VoiceSecure source code
- Pre-trained models
- Scripts for evaluation, metrics computation, and figure generation
- Instructions for training on custom datasets
- Precomputed results used in the paper
- A detailed Readme to guide setup and execution

A.2.3 Hardware dependencies

None. The artifact is designed to run on standard consumer-grade hardware. No GPU is required. All evaluations can be performed using a CPU machine with at least 8 GB of RAM.

A.2.4 Software dependencies

The artifact requires:

- **Python 3.8**, with packages listed in the provided requirements.txt
- **MATLAB R2021a or later**, with the following toolboxes: Audio, Signal Processing, and 5G Toolboxes.

A.2.5 Benchmarks

The following datasets and models were used in the experiments presented in the paper:

- **Speech Datasets:** Open-source corpora including LibriSpeech, VoxCeleb, CommonVoice, and VCTK.

- **User Study Responses:** Collected from human listeners to evaluate perceptual intelligibility using STOI.
- **Speech Recognition:** Whisper, DeepSpeech, and Wav2Vec2.
- **Speaker Verification:** x-vector, ECAPA-TDNN, and i-vector models.

All benchmark data (except raw VoxCeleb audio due to licensing) and model outputs—including speaker embeddings, mismatch rates (MMR), WER, and STOI scores—are included in the artifact package. Pre-trained speaker verification models are bundled with the artifact. Open-source speech recognition models are supported and installed via dependencies specified in **requirements.txt**.

A.3 Set-up

This section provides detailed steps to install and configure the environment necessary for evaluating the VoiceSecure artifact. Following the instructions below, evaluators will be able to run a basic test to verify the correct installation and functionality of all required components. Please note that the full setup process may take approximately 2 hours.

A.3.1 Installation

1. **Download the artifact files** from Zenodo: https://doi.org/10.5281/zenodo.15603263 The download includes two zip files: **Data2.zip (23 GB)**, and **VoiceSecure_Artifacts_Scripts.zip (1.5 GB)**.
2. **Unzip both files into a common directory**. Your directory structure will look like:
   - Data2/
   - ScriptForApplyingVoiceSecure/
   - ScriptForTrainingModel/
   - ScriptForComputingMetrics/
   - ScriptsForCompiledResults/
   - ScriptForDataSetCreation/
   - Trained_Model/
   - Testing_Installation/
   - requirements.txt
   - README.md
3. **Install Miniconda (if not already installed):**
   bash Miniconda3-latest-Linux-x86_64.sh
   source /miniconda3/bin/activate
4. **Create and activate a conda environment:**
   conda create –name py38 python=3.8.18
   conda activate py38
5. **Install Python dependencies:**
   pip install -r requirements.txt
6. **MATLAB Dependencies:** MATLAB (R2021a or later) with the Audio, Signal Processing, and 5G toolboxes.
7. **To run DeepSpeech-based evaluations:**
   - Clone from the GitHub: https://github.com/SeanNaren/deepspeech.pytorch
   - Navigate to the repo
   - pip install -r requirements.txt
   - Ensure this repo is added to your environment path.

A.3.2 Basic Test

To verify that the installation has been completed successfully, we provide a testing script located in the `Testing_Installation/` directory. This script performs checks to ensure that all major components of the system are functional, including the VoiceSecure model, the speaker embedding models (x-vector, ECAPA), and the ASR models used for computing word error rates (Whisper, DeepSpeech, Wav2Vec2). To run the test, execute the script `TestPythonInstallation.py`. If the setup is correct, the script will print seven confirmation messages, each beginning with the word `"Functional"`, corresponding to the various components being tested. At the end, it will display the message `"Installation complete"`, indicating that all necessary modules are working as expected.

A.4 Evaluation workflow

VoiceSecure is a speech transformation method designed to obfuscate speaker identity and speech content while preserving intelligibility for human listeners. To evaluate its effectiveness, we perform three key types of evaluation:

1. **Speaker Verification:** We use three state-of-the-art speaker verification models (X-Vector, ECAPA-TDNN, and i-Vector) to extract speaker embeddings from both the original and VoiceSecure-modified speech. We then compute the speaker mismatch rate (MMR), which quantifies how often the modified speech is misidentified as originating from a different speaker. A **higher mismatch rate** indicates stronger anonymization, which is the desired outcome.
2. **Word Error Rate (WER):** To assess the impact of VoiceSecure on speech recognition accuracy, we evaluate the modified speech using three automatic speech recognition (ASR) models: Whisper, DeepSpeech, and Wav2Vec2. We compute the WER for both original and modified speech. A **higher WER** reflects greater disruption to ASR systems, which is the intended effect.

3. **Speech Intelligibility:** We measure intelligibility using the Short-Time Objective Intelligibility (STOI) metric, which correlates strongly with human perceptual scores. In this context, a **higher STOI score** is desirable, as it indicates that the modified speech remains understandable to human listeners despite the applied transformations.

Our artifact includes all necessary scripts to apply VoiceSecure modifications, compute speaker embeddings, mismatch rate, WER, and STOI scores. It also provides original and transformed speech samples, along with pre-computed embeddings, mismatch rates, WERs, and STOI scores. In addition, we include MATLAB scripts that generate the key figures and tables from the paper using the pre-evaluated data.

A.4.1 Major Claims

(C1): VoiceSecure achieves a 52% Word Error Rate, 33% Speaker Mismatch Rate, and 72% intelligibility, demonstrating its ability to protect privacy while preserving human understanding. (Section 9)

(C2): Compared to existing baselines, VoiceSecure offers a better trade-off between privacy and intelligibility, outperforming McAdams in both speaker anonymization and ASR obfuscation, and exceeding VoiceMask by 12% in intelligibility. (Section 9.1, Figure 9)

(C3): Subjective evaluations confirm that VoiceSecure maintains perceived speech clarity while enhancing privacy, making it suitable for real-world deployment. (Section 9.4, Figure 11).

A.4.2 Experiments

(E1): [60 human-minutes + 20 compute-hours] This experiment evaluates VoiceSecure's effectiveness in anonymizing speaker identity and obfuscating speech content, supporting major claim C1.
Preparation: Ensure all dependencies are installed. Confirm the availability of the pre-trained VoiceSecure model and evaluation datasets.
Execution: Run the scripts in `ScriptForApplyingVoiceSecure/` to apply the transformation and generate modified speech samples. Then execute `ComputeSpeakerEmbeddings.py` and `ComputeWER.py` to compute and store speaker embeddings and word error rates, respectively. For this experiment, we recommend using the LibriSpeech dataset along with the X-Vector and DeepSpeech for speaker embeddings and word error rates, respectively. As this process is computationally intensive, we also provide pre-modified speech samples as well as pre-computed embeddings and word error rates. Finally, run the scripts in `ScriptForComputingMetrics/` to compute the speaker mismatch rate, mean word error rate, and speech intelligibility scores.
Results: The experiment outputs speaker mismatch rates, word error rates, and intelligibility scores, supporting claim 1.

(E2): [60 human-minutes + 30 compute-hours] This experiment supports major claim C2 by comparing VoiceSecure against baselines across various state-of-the-art speaker verification and speech recognition systems in terms of privacy protection and intelligibility.
Preparation: Ensure `Data2/LibriSpeech_Dev/` contains speaker embeddings (for three models), word error rates (for three models), and intelligibility scores for both original and processed speech samples (noise, McAdams, VoiceMask, and VoiceSecure).
Execution: Use the scripts in 'ScriptForComputingMetrics/' to compute and store MMR, WER, and STOI for all methods. Then run the 'ScriptsForCompiledResults/Plotting_Compiled_Results' MATLAB script to generate comparison figures.
Results: This reproduces Figure 9 in the paper, demonstrating that VoiceSecure achieves a superior trade-off between privacy and intelligibility compared to existing methods.

(E3): [10 human-minutes + 0.1 compute-hour] This experiment supports major claim C3 by analyzing listener feedback on perceived speech intelligibility after transformation.
Preparation: Navigate to `Data2/UserStudy/` and verify that the listener response data is present.
Execution: Run the provided MATLAB script 'ScriptsForCompiledResults/Plotting_UserStudy_Results' to process user study responses and generate aggregated perceptual scores.
Results: The experiment reproduces the user study results (Figure 11) in the paper, validating that VoiceSecure maintains high perceived intelligibility while offering strong privacy protections.

A.5 Notes on Reusability

The scripts in `ScriptForApplyingVoiceSecure/` allow users to apply VoiceSecure-based modifications to any speech dataset, enabling privacy protection across diverse use cases. Additionally, the script in `ScriptForTrainingModel/` enables training the model from scratch on custom datasets, allowing adaptation to different domains or data conditions.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: McSee: Evaluating Advanced Rowhammer Attacks and Defenses via Automated DRAM Traffic Analysis

Patrick Jattke, ETH Zurich
Michele Marazzi, ETH Zurich
Flavien Solt, UC Berkeley
Max Wipfli, ETH Zurich
Stefan Gloor, ETH Zurich
Kaveh Razavi, ETH Zurich

A Artifact Appendix

A.1 Abstract

McSee is an open-source platform for capturing and analyzing DRAM bus traffic. It is based on a high-frequency oscilloscope with a custom-designed interposer and an optimized trace data processing pipeline, including a DDR4/5 decoder.

The artifacts of our work include the McSee platform, including the design of the custom-built DDR5 UDIMM interposer and the oscilloscope decoding and analysis pipeline. Further, we provide the code of our experiments and the captured oscilloscope traces for validating our claims.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our work does not involve any end-to-end Rowhammer attacks, neither does McSee capture any data signals from the DRAM bus. Therefore, our artifacts do not raise any security, privacy, or ethical concerns.

A.2.2 How to access

The artifacts and collected data is available on Zenodo at `https://doi.org/10.5281/zenodo.15610915`. To facilitate the access, we also publish the artifacts (code only) on GitHub at `https://github.com/comsec-group/mcsee`.

A.2.3 Hardware dependencies

Reproducing our results requires systems with specific CPUs, DDR5 UDIMMs requiring RFM, and the hardware of the McSee platform.

Systems. We used following CPUs for our experiments.

- **DDR5 experiments**:
  - Intel Alder Lake (i7-12700K), ucode: `0x3a`
  - Intel Raptor Lake (i7-13700K), ucode: `0x129`
  - AMD Zen 4 (Ryzen 7 7700X), ucode: `0xa601203`
- **DDR4 experiments**:
  - Intel Coffee Lake (i7-8700K), ucode: `0xf8`

DDR5 UDIMMs. Validating our results on real hardware requires DDR5 UDIMMs where the RFM required bit is set and valid RFM values are provided in the SPD data. This data can be obtained from the SPD EEPROM of the UDIMM by using the SPD reader tool provided in the artifacts.

Reproducing the DDR5 bit flips that we found on one of the devices (S_4, Samsung), requires the same DDR5 UDIMM as used in Zenhammer: a 16 GB Samsung UDIMM, model `M323R2GA3BB0-CQKOD` with 4800 MHz.

McSee platform hardware. The McSee platform is based on the following hardware components (see §4.1 for details):

- Oscilloscope (TELEDYNE SDA 806Zi-B)
- Digitizer (TELEDYNE HDA125-18-LBUS)
- Differential probe (TELEDYNE DH08-PB2)
- Analog probe (TELEDYNE PP021)
- Solder-in leads (TELEDYNE HDA-DLS-18QL)
- DDR5 UDIMM interposer (custom design, see artifacts)
- 5 GbE USB NIC (Startech US5GA30)

We provide the oscilloscope captures to reproduce our results such that no access to McSee and the exact same hardware is required.

A.2.4 Software dependencies

We require a Linux-based operating system to run our software. Our experiments were conducted on Ubuntu 20.04.6/22.04.5/24.04.1 LTS and GNU/Linux kernel versions 6.1.118/6.8.0/6.9.0. However, our experiments and tools do not rely on any particular OS kernel version.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

We refer to the README file in the repository for installation instructions.

A.3.2 Basic Test

The README of the respective component of our McSee platform contains information on how to check if it has been installed correctly.

A.4 Evaluation workflow

A.4.1 Major Claims

Our work makes the following major claims:

(C1): The **McSee data processing pipeline** is open source and allows to efficiently capture and decode DDR4/5 bus traffic (see §4.2). We describe where to find the different software components of McSee in the README of our artifacts repository.

(C2): The **PCB design** of our custom-built DDR5 UDIMM interposer as part of the McSee platform is open source (see §4.1). The design can be found in the `ddr5-udimm-interposer-pcb` directory of the artifacts repository.

(C3): The **DDR5 SPD decoder** is open source and allows to decode the RFM values of DDR5 UDIMMs (see §6.1). The decoder can be found in the `spd-decoder` directory of the artifacts repository. We describe in experiment (E1) how we used the decoder to extract the relevant values.

(C4): We show that **Sledgehammer's** activation throughput declines when hammering more than six banks in parallel (see §5.1). This is proven by experiment (E2) whose result is illustrated in Figure 7.

(C5): We show that **Sledgehammer** does not achieve less reordering of aggressor accesses when hammering more banks in parallel (see §5.1). This is proven by experiment (E3) whose result is illustrated in Figure 8.

(C6): We show that **RowPress'** average row-open time on a real system is in the range where the reduction of the minimum activation count (AC_{min}) to trigger a bit flip is small (see §5.2). This is proven by experiment (E4) whose result is illustrated in Figure 9.

(C7): **Systematic bit flipping** allows to recover the DRAM addressing functions of our Zen 4 CPUs (see §6.2). This is proven by experiment (E5) whose result is presented in Table 3.

(C8): Intel Raptor Lake CPUs do not issue any **RFM** commands, but they employ a memory controller-based mitigation **`pTRR`** that refreshes aggressor-adjacent rows (see §6.3). This is proven by experiment (E6) whose result is illustrated in Figure 10.

(C9): Intel **pTRR** targets victim rows with a probability of 0.091% (see §6.4). This is proven by experiment (E7) whose result is illustrated in Figure 11.

(C10): Given the pTRR probability, we calculate the **pTRR bypass time**: devices protected with pTRR with a Rowhammer threshold of 13200, 16700, and 18800 activations can be bypassed with roughly 50% attack success probability in less than an hour, one day, and one week, respectively. This is proven by experiment (E8) whose result is illustrated in Figure 12.

A.4.2 Experiments

As we require multiple different systems attached to the McSee platform to reproduce our results, we provide the (unprocessed) oscilloscope traces of the following experiments. We report effort as a triple "[HM/CH/DU]" of human-minutes (HM), compute-hours (CH), and disk usage in gigabytes (DU).

(E1): **RFM values** [1:30 h / 0:20 h / 1 MiB]. We use our DDR5 SPD decoder to extract the RFM values from the SPD data of the 29 DDR5 UDIMMs that we use in our experiments.

Preparation: Follow the instructions in the README file in `experiments/e1-rfm-values`.

Execution: Extract the SPD data of the DIMMs, then parse the `arfm` object of the generated JSON files. We provide the collected SPD data in the experiment's `data/` directory.

Results: As shown in Tbl. 7, 19 of 30 devices have valid RFM values (i.e., `raaimt` and `raammt` are not set to `RFU`). One of the 19 devices has the RFM required bit (`rfm_req`) set.

(E2): **Sledgehammer's activation throughput** [2:20 h / 1h / 40 GiB]. Using McSee, we measure the per-bank and total activation throughput of Sledgehammer.

Preparation: Follow the instructions in the README file in `experiments/e2-sledgehammer`.

Execution: Run our modified Sledgehammer fork on the Coffee Lake system while taking three captures of each 1 ms with the McSee platform. Repeat this for 1 to 16 banks.

Results: The activation rate per bank strongly declines when hammering more than six banks in parallel, as shown in Fig. 7.

(E3): Sledgehammer's access reordering [0:05 h / 0:01 h / 40 GiB]. We analyze the distance (in activations) between consecutive accesses to the same aggressor in Sledgehammer.

Preparation: Follow the instructions in the README file in `experiments/e2-sledgehammer`.

Execution: No execution is required, we use the data gathered in (E2) and use a script to analyze the traces.

Results: Increasing the number of in parallel hammered banks, increases the frequency of reordering aggressor accesses, as shown in Fig. 8.

(E4): RowPress' row-open time [1h / 0:40 h / 20 GiB]. We run the real system demonstration of RowPress on a Coffee Lake system to measure the average row-open time using McSee.

Preparation: Follow the instructions in the README file in `experiments/e4-rowpress`.

Execution: Run our modified RowPress fork on the Coffee Lake system while taking two captures of each 1 ms with the McSee platform. Repeat this for 1, 4, 8, 16, 32, 64, 80, and 128 column reads.

Results: An average t_{AggON} time between approx. 50 ns (1 cache block read) and almost 300 ns (128 cache block reads) as shown in Fig. 9.

(E5): Systematic bit flipping [0:20 h / 0:30 h / 30 GiB]. We run the systematic bit flipping experiment on our Intel Raptor Lake system to recover the DRAM addressing functions and labels.

Preparation: Follow the instructions in the README file in `experiments/e5-systematic-bit-flipping`.

Execution: Run the provided code implementing the systematic bit flipping experiment described in Algorithm 1. This code automatically triggers Run the code on the three different DRAM configurations listed in Tbl. 3.

Results: The three different sets of DRAM addressing functions as shown in Tbl. 3.

(E6): Existence of pTRR [0:20 h / 0:15 h / 100 MiB]. We verify that on Intel Raptor Lake CPUs, the memory controller infrequently issues an activate command to a row neighboring our hammered aggressor row.

Preparation: Follow the instructions in the README file in `experiments/e6-ptrr-existence`.

Execution: We use the same hammering workload with a double-sided aggressor pair as in the pTRR probability experiment (E7).

Results: The aggressor-adjacent rows are refreshed by pTRR, as shown in Fig. 10, although they are never accessed in the experiment.

(E7): pTRR probability [0:30 h / 8h / 150 GiB] We run a double-sided hammering workload on our Intel Raptor Lake system to determine the probability of pTRR to targeting the victim row.

Preparation: Follow the instructions in the README file in `experiments/e7-ptrr-probability`.

Execution: We hammer a double-sided aggressor pair while capturing traces with the McSee platform.

Results: A bar plot fitting a binomial probability distribution with $p = 0.00091$ as shown in Fig. 11.

(E8): pTRR attack success rate [0:05 h / 0:05 h / 10 MiB] We use the determined pTRR probability to calculate the success rate of bypassing pTRR over time.

Execution: Run the provided MATLAB script `calculate_prob.mlx` either locally or using the MATLAB cloud service.

Results: A plot as in Fig. 12 showing that devices protected with pTRR with a Rowhammer threshold of 13200, 16700, and 18800 activations can be bypassed with roughly 50% attack success probability in less than an hour, one day, and one week, respectively.

A.5 Notes on Reusability

The core of our McSee platform is the DDR4/5 DRAM command decoder. As this decoder works on the oscilloscope traces that have been beforehand transformed into the CSV format, it is platform-agnostic and can be used with any other hardware setup that is capable of producing such CSV traces.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Not so Refreshing: Attacking GPUs using RFM Rowhammer Mitigation

Ravan Nazaraliyev[*1], Yicheng Zhang[1], Sankha Baran Dutta[2], Andres Marquez[3], Kevin Barker[3], Nael Abu-Ghazaleh[†1]

[1]*University of California, Riverside*
[2]*Brookhaven National Laboratory*
[3]*Pacific Northwest National Laboratory*

A Artifact Appendix

This Artifact Appendix provides the necessary details to enable evaluations of our artifacts.

A.1 Abstract

Ou artifact is composed of one main part, code scripts for GPU covert, side channel and slow-down attacks, and an additional part for machine learning classification algorithms along with pre-collected data.

A.2 Description & Requirements

We have provided code scripts to recreate covert channel and slow-down attacks locally. We also provide the side channel code that we used to collect data along with pre-collected data and classification models for fingerprinting attacks.

A.2.1 Security, privacy, and ethical concerns

Our codes do not cause any issues regarding security, privacy, and ethics. The provided artifact should be run locally on a machine, and the codes show proof-of-concept attacks. The codes, as expected, do not cause any harm to the machine nor steal private information.

A.2.2 How to access

We have provided our artifacts in Zenodo repository. The main part is located in notsorefreshing_artifact repository. When you enter the repository, you will see the folder **`notsorefreshing_artifact`** folder which is the main folder that contains code scripts. The other folders contain ML classification models for collected data. We provide a README file for main code scripts inside `notsorefreshing_artifact` folder. We provide an additional README file (**`ReadMe_For_5_side_channel_attacks.md`**).

A.2.3 Hardware dependencies

Our code scripts require a custom CPU (our machine is Intel Xeon E5-1620). The provided experiments require NVIDIA GeForce RTX 4060 and Jetson Orin AGX LPDDR5. If you do not intend to collect data on Jetson or do not have access to one, you can use the already provided dataset.

A.2.4 Software dependencies

The evaluation machine should have NVIDIA Driver and CUDA toolkit installed. We have provided the driver as part of artifact. Please refer to CUDA toolkit download for CUDA toolkit installation. To run Blender object fingerprinting, the machine needs to have the Blender package, and you can install Blender on Linux machine using *snap install blender –classic*. Please refer to the README file for further information.

A.2.5 Benchmarks

For all side channel experiments, we have provided necessary benchmarks as part of the artifact. We only do not provide Blender benchmark for slow-down attack as it is easy to get it from Blender benchmark. To run this benchmark, you have to have Blender package (refer to Sec A.2.4) and git clone Phoronix Test Suite.

A.3 Set-up

We do not require a specific setup. The attack codes (CUDA) need to be run with GUI mode off. You can disable GUI using `sudo service lightdm stop`. If your machine does not have `lightdm`, you can easily install it using `sudo apt install lightdm`. Please refer to `README` for further details.

[*]rnaza005@ucr.edu
[†]naelag@ucr.edu

A.3.1 Installation

Please refer to Sections A.2.2 and A.2.4 for related information. Please refer to README for further details.

A.3.2 Basic Test

The covert channel attack consists of Python scripts, as described in the README file. The test outputs the achieved bandwidth and error rate. For the slow-down attack, run the Blender benchmark with and without the attack active. When the attacker is active, the benchmark takes longer, and the ratio of execution times gives the slow-down factor. The side-channel scripts begin collecting data when executed. If pre-collected data suffices for demonstrating functionality or reproducibility, data collection is not required, as the necessary traces are already included.

A.4 Evaluation workflow

We provide the necessary details for running the experiments in the README files. Briefly, for the covert channel, run `RFM_Covertchannel_with_physidech.py` to send a random message, then run `covertchannelevaluation.py` to measure the error rate. This covert channel works on GeForce RTX 4060. Start MPS before running this attack (`nvidia-cuda-mps-control -d`). This artifact corresponds to Section 4.

Slowdown attack. Run `./slow_down` to start the slow-down code, then launch the benchmark. We used the Blender benchmark; see Section A.4. Start MPS before this attack (`nvidia-cuda-mps-control -d`). This artifact corresponds to Section 7.

Side-channel codes:

GeForce RTX 4060 (with MPS). For Blender 3D object fingerprinting, run `sidechannelautomatic.py`. It launches the memory profiler, renders Blender objects, collects samples, runs `memorygram.py`, and deletes CSV files to save space. Requirements: 1) install Blender (Section A.2.4), 2) start MPS (`nvidia-cuda-mps-control -d`). This artifact corresponds to Section 5.

Jetson Orin AGX 64GB (without MPS). Go to the `AttacksOnJetson` folder. The three attacks are in separate folders. For SPEC fingerprinting, run `./specfingerprint.sh`. For Web and Video attacks, run `./timeseriesdatacollection_web` and `./timeseriesdatacollection_video`, respectively. Data is saved in CSV files. This artifact corresponds to Section 6.

A.4.1 Major Claims

Our artifacts provide reproducible and functional proof-of-concept code implementations for the attacks presented in the paper.

(C1): *We demonstrate a covert channel across two GPU processes based on RFM Rowhammer mitigation. The channel achieves high bandwidth with a low error rate. Results are shown in Table 1.*

(C2): *We demonstrate side-channel fingerprinting attacks on GDDR and LPDDR systems. On GeForce RTX GPUs, we achieve over 90% precision in identifying the victim application (Attack1) or Blender character (Attack2) rendered on a shared GPU (Tables 2 and 3). On Jetson Orin AGX (LPDDR), we show that a spy process can capture RFM fingerprints of a CPU process. We demonstrate application, web, and video fingerprinting attacks (Attack3–5), each achieving near or above 90% accuracy (Table 4).*

(C3): *We show that an attacker can significantly slow down another application by exploiting RFM-based Rowhammer mitigation, with an average slowdown of 4.8×.*

A.4.2 Experiments

We have already provided the necessary details on how to run our code to reproduce results. Below, we provide explicit instructions corresponding to claims in the previous section.

(E1): *[Covert channel] [a couple of minutes of human time]:*
How to: Start MPS using the `nvidia-cuda-mps-control -d` command. In the main artifact folder, run `RFM_Covertchannel_with_physidech.py` to initiate covert communication. After completion, it will output status messages and the channel bandwidth. Then, run `covertchannelevaluation.py` to obtain the error rate.
Preparation: Disable GUI using `sudo service lightdm stop`. If `lightdm` is not installed, install it via `sudo apt install lightdm`.
Execution: Run the mentioned Python scripts to obtain results.
Results: You should observe bandwidth and error rate values matching or close to those reported in the paper.

(E2): *[Side channel] [>1 day for full data collection]:*
How to: We provide scripts for both data collection and classification. Full data collection takes more than a day per script (4 total). For convenience, we suggest running ML classifiers on pre-collected data, which was gathered using the same scripts.
Preparation: For data collection, disable GUI using `sudo service lightdm stop`. No preparation is needed to run the ML classifiers.
Execution: Follow the README instructions to run data collection and analysis scripts.
Results: Data collection alone does not yield results. Run ML scripts to obtain F1 score, precision, and recall.

(E3): *[Slow-down attack] [2–3 hours]:*
How to: Run the Blender benchmark with and without

the attacker to observe slowdown.

Preparation: Disable GUI via `sudo service lightdm stop`. If `lightdm` is not installed, use `sudo apt install lightdm`.

Execution: The Blender benchmark is available at OpenBenchmarking. To run it, install Blender (see Section A.2.4) and clone Phoronix Test Suite. To launch the attacker, go to the main artifact folder and run `./slow-down`.

Results: You should observe increased benchmark runtime when run concurrently with the attacker.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Posthammer: Pervasive Browser-based Rowhammer Attacks with Postponed Refresh Commands

Finn de Ridder
ETH Zurich

Patrick Jattke
ETH Zurich

Kaveh Razavi
ETH Zurich

A Artifact Appendix

A.1 Abstract

Posthammer shows that browser-based Rowhammer attacks are pervasive. In particular, we show that the majority of DDR4 devices are vulnerable to *clflush*-free Rowhammer patterns that the attacker can launch from client-side JavaScript.

Posthammer includes the following artifacts: first, an experiment for inducing and measuring refresh postponement. Second, a fuzzer for triggering Rowhammer bit flips natively using *clflush*-free and *refresh-postponing* patterns in a large search space. Third, a JavaScript version of the fuzzer that searches for effective patterns in a reduced search space. Fourth, an exploit that obtains an arbitrary read-write primitive in the address space of the JavaScript process.

A.2 Description & Requirements

To reproduce our results, the evaluator needs:

1. The hard- and software dependencies listed under A.2.3 and A.2.4.
2. The artifacts themselves, which are available at `https://doi.org/10.5281/zenodo.14738152` and `https://github.com/comsec-group/posthammer` (preferred).

After unpacking the artifacts, execute the following commands:

1. For the first two artifacts, the experiment and native fuzzer, in `./native-fuzzer`, execute `./main.sh |& tee dump`. Simply run it again if you get an error about `mem.o` missing. The refresh postponement experiment requires the `SPLIT_DETECT` macro to be defined.
2. For the JavaScript fuzzer, in `./js-dbg-hugepages`, execute `make clean && make; make`. We run `make` twice to force execution despite the transpiler warnings.
3. For the exploit, in `./js-exploit`, also execute `make clean && make; make`.

The exploit may fail either due to a segfault or an assertion failing (e.g. because it cannot find an eviction set). In these cases, please try again. Similarly, the native fuzzer may fail to find its first eviction set, but should otherwise not crash.

The exploit will run until it has (intentionally) segfaulted at `0x1337` while the native fuzzer will fuzz indefinitely while writing its results to `./pattern/flip.csv`. Human-friendly output can be found in `dump` (if captured as suggested above). Search for `->` to view the bit flips that have been triggered.

A.2.1 Security, privacy, and ethical concerns

The artifacts are safe. As mentioned above, upon successful execution, the exploit causes a harmless segmentation fault at address `0x1337`. This segmentation fault will be reported in the kernel log, see `dmesg`. Unsuccessful runs will trigger arbitrary but equally benign segmentation faults in the address space of the script.

A.2.2 How to access

The latest version of the artifacts is available at `https://github.com/comsec-group/posthammer`. The directory structure suggests where which artifact can be found: the exploit is contained in `js-exploit`, the native fuzzer in `native-fuzzer`, and the JavaScript fuzzer in `js-dbg-hugepages`.

A.2.3 Hardware dependencies

1. A desktop machine with an Intel Core i7-7700K (Kaby Lake) processor. *This is a strict requirement for the exploit.* The native fuzzer also works on Intel Core i7-8700K (Coffee Lake) CPUs.
2. A (single) vulnerable DDR4 DIMM. The specifics of the DIMMs used in the paper are given in the paper's appendix. For testing, we recommend a Samsung (A) DIMM, as they are most vulnerable to Posthammer.

A.2.4 Software dependencies

1. A Debian-based operating system. We have used Ubuntu 18.04.6 under Linux 5.4.0-150-generic. Although we recommend to use exactly these versions, newer ones might also work. For example, the native fuzzer also works on Ubuntu 22.04.5 under Linux 5.15.0-130-generic.
2. Certain software packages. The details are given in the README. The installation should be straightforward.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

We refer the evaluator to the README for the installation instructions.

A.3.2 Basic Test

The following tests can be used to verify the environment.

1. The availability and version of the TypeScript transpiler:

```
$ tsc --version
Version 2.7.2
```

2. The availability and version of the JavaScript shell:

```
$ ./jsshell-130/js --version
JavaScript-C130.0
```

A.4 Evaluation workflow

A.4.1 Major Claims

C0: Rowhammer patterns crafted as specified in the paper (Sections 5 and 6) induce refresh postponement. This is proven by Figure 2 of the paper, which can be reproduced using E0, see below.

C1: On the majority of DDR4 devices, these non-uniform and/or refresh postponing *clflush*-free Rowhammer patterns trigger bit flips while the self-evicting patterns used in previous work do not. This is proven by Experiment 6 (Table 2) in the paper.

C2: It is possible to trigger these bit flips from JavaScript with a reduced search space. See also Experiment 6 and Table 2.

C3: The bit flips triggered by these patterns can be used to obtain an arbitrary read-write primitive in the JavaScript runtime.

A.4.2 Experiments

The experiments below map linearly to the claims in A.4.1.

E0: *Refresh postponement*: produces Figure 2 of the paper and therefore shows that our patterns induce refresh postponement.

1. Navigate to `./native-fuzzer/pattern/pattern.c`. Open the file and enable the `SPLIT_DETECT` macro.
2. Execute `./native-fuzzer/main.sh`. This should take around 30 minutes. Plot the data it has written to `./native-fuzzer/split.csv`.

E1: *Native fuzzer:* explores the *clflush*-free, non-uniform, and refresh-postponing pattern space. Triggers bit flips on most DDR4 devices, see again Table 2.

1. Make sure the `SPLIT_DETECT` macro is *undefined* (default).
2. Execute `./native-fuzzer/main.sh |& tee dump`. Depending on the vulnerability of the DIMM, it may take several hours until the first bit flip. As Table 2 shows, however, for most DIMMs, 6 hours should suffice.

As mentioned A.2, we recommend piping the output to a file and grepping it for the arrow symbol `->` to check for bit flips.

E2: *JavaScript fuzzer*: shows that the native patterns translate to JavaScript. Relies on huge pages for convenience.

1. Navigate to `./js-dbg-hugepages` and execute `make clean && make; make`.

The `Makefile` should automatically enable transparent huge pages (THPs).

E3: *Exploit*: the exploit. Does *not* rely on huge pages. Uses two bit flips to obtain an arbitrary read-write primitive in the JavaScript runtime. To showcase the primitive, we write to virtual address `0x1337` and segfault.

1. Navigate to `./js-exploit` and execute `make clean && make; make`.

The exploit may take up to an hour to complete. Moreover, it may segfault before completion due to *unwanted* bit flips. It will print `About to segfault at 0x1337...` just before segfaulting as planned, which may be verified by inspecting `dmesg`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Relocate-Vote: Using Sparsity Information to Exploit Ciphertext Side-Channels

Yuqin Yan[†], Wei Huang[†‡], Ilya Grishchenko[†], Gururaj Saileshwar[†], Aastha Mehta[*], and David Lie[†]

[†]University of Toronto, [‡]Seneca Polytechnic, [*]University of British Columbia
yuqin.yan@mail.utoronto.ca, wei.huang1@senecapolytechnic.ca, gururaj@cs.toronto.edu,
aasthakm@cs.ubc.ca, {ilya.grishchenko,david.lie}@utoronto.ca

A Artifact Appendix

A.1 Abstract

This artifact includes the source code, scripts, and datasets used to support the main claims, figures, and tables presented in the paper. The paper demonstrates how a malicious hypervisor can infer sparsity patterns in the memory of victim confidential Virtual Machines (CVMs) protected by AMD SEV-SNP. This is accomplished by identifying encrypted memory blocks that contain either prevalent or non-prevalent values, exploiting ciphertext side channels via the `SNP_PAGE_MOVE` command—designed initially for resource management.

At a high level, the artifact provides: (1) a malicious hypervisor implemented as part of a modified Linux kernel, (2) code simulating the behavior of a victim CVM, (3) monitoring logic used by the hypervisor to observe the victim through controlled channels and ciphertext side channels, and (4) analysis tools to recover and evaluate sparsity information from the victim's memory.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact is based on a modified kernel implementation, SEV-Step. Both the original version and our implementation prioritize the functionality of attack primitives over kernel stability. As a result, the system may experience unpredictable behavior, including kernel crashes, instability, and unresponsiveness. **WARNING:** This artifact is intended strictly for research and evaluation purposes. **Do not use it in production environments.**

Known limitations and usage guidelines:

- Do not run multiple VMs simultaneously.
- Do not assign multiple cores to a single VM, which may prevent the VM from booting.
- Avoid forcefully terminating an evaluation program, which may cause system issues.
- The CVMs cannot be correctly shut down sometimes. Specifically, you cannot terminate a running CVM by either Ctrl-C in the terminal with the "login" prompt or `pkill`. If this happens, you need to reboot the host machine.

A.2.2 How to access

The artifact is available at Zenodo via the following concept DOI: https://doi.org/10.5281/zenodo.15609905.

A.2.3 Hardware dependencies

The evaluation of the attack primitive leveraging the `SNP_PAGE_MOVE` command is conducted on AMD SEV-SNP platforms. Since AMD fixed an implementation bug in firmware version 1.55 (build 20), our evaluation requires firmware version 1.55.20 or later for efficient page relocation.

A.2.4 Software dependencies

The host platform is running Ubuntu 24.04.1 LTS.
Environments supporting `pip install`. Examples of such environments are Anaconda and `python-venv`. Recently, `pip install` in the native, raw environment was prohibited due to PEP 668—Marking Python base environments as "'externally managed".
Blender. Blender is used to manually assemble and visualize the pieces leaked from the OpenVDB operations performed by the victim. The installation instructions can be found at https://www.blender.org/download/.

A.2.5 Benchmarks

Dataset. We rely on the external dataset, CQ500. Although it is externally available at https://www.kaggle.com/datasets/crawford/qureai-headct, we have included the necessary components in our artifact package under `openvdb-leak/data/qureai-headct/` to ensure long-

term accessibility. The distribution complies with the original license, CC BY-NC-SA 4.0. Both the license file (`LICENSE`) and a corresponding `README.md` are provided in the directory.

Weights of Large Language Model (LLM). Our experiments relied on the weights of `ReLULlama-13b`, available at https://huggingface.co/PowerInfer/ReluLLaMA-13B-PowerInfer-GGUF. For the ease of reproducibility, we include the weights in the artifact package. The weights are released under the `llama2` license, and the license files `LICENSE.txt` and `Notice` are included in the package.

A.3 Set-up

A.3.1 Installation

The installation consists of four steps. Follow the detailed instructions in the "Installation" section of `README.md`.

1. Set `RV_ROOT_DIR` to the root directory of the artifact by executing the script in it:

```
source ./env.sh
```

2. Create the CVM images, `aslr.qcow2` and `big-disk.qcow2` (Table 1) in `$RV_ROOT_DIR/vm-images-creation/`. The first line takes around 10 minutes to execute. The second line can take around 30 minutes. The default username is `ubuntu` and the password is `123456`.

```
$RV_ROOT_DIR/vm-images-creation/create_vm1.sh
$RV_ROOT_DIR/vm-images-creation/create_vm2.sh
```

3. Build and install the customized Linux kernel.

```
cd $RV_ROOT_DIR/sev-step
./build.sh ovmf && \
./build.sh qemu && \
./build.sh kernel
```

The above commands should generate `kernel-packages/` in `$RV_ROOT_DIR/sev-step/`. Install the `*.deb` packages using the following command:

```
sudo dpkg -i \
    $RV_ROOT_DIR/sev-step/kernel-packages/*.deb
```

After installation, execute `sudo update-grub` to see if the following lines are in the output:

```
Found linux image: /boot/vmlinuz-5.19.0-rc6-sev-step-999ae99
Found initrd image: /boot/initrd.img-5.19.0-rc6-sev-step-999
    ae99
```

If so, the installation should be successful. Record its index in the output, where the index starts from 0. For example, if the output is:

```
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-6.8.0-49-generic
Found initrd image: /boot/initrd.img-6.8.0-49-generic
Found linux image: /boot/vmlinuz-5.19.0-rc6-sev-step-999ae99
Found initrd image: /boot/initrd.img-5.19.0-rc6-sev-step-999
    ae99
...
```

The index of the customized kernel is 2 (the third line, starting from 0). Now update the `GRUB_DEFAULT` in `/etc/default/grub` to boot into the new kernel by default.

```
GRUB_DEFAULT="1>[index]"
```

Here, `[index]` is the index of the new kernel in the GRUB menu just recorded. Then update GRUB configuration:

```
sudo update-grub
```

Reboot the machine and verify the kernel version with `uname -r`. It should output `5.19.0-rc6-sev-step-999ae99`.

4. Compile the user-space projects for the experiments.

```
cd $RV_ROOT_DIR && make
```

A.3.2 Basic Test

Launch guest CVMs. Run the following command to launch **VM1** (Table 1):

```
$RV_ROOT_DIR/launch-vm1.sh
```

Upon a successful launch, a `login` prompt should appear in the terminal (pressing "Enter" is required sometimes). If the launch fails, check the script's output for potential troubleshooting instructions. Rebooting the host machine to reset the state if necessary.

For launching **VM2**, execute the following instruction in a new terminal (`source ./env.sh` first in it), which also kills **VM1** first due to the implementation limitation that **VM1** and **VM2** cannot be run simultaneously. When switching VMs, read the script output carefully for instructions and reboot the host machine if necessary.

```
$RV_ROOT_DIR/launch-vm2.sh
```

VM Name	`qcow2` image	`memoy-size` (in MB)
VM1	`aslr.qcow2`	4096
VM2	`big-disk.qcow2`	4096

Table 1: CVMs used in the evaluation.

Test attack primitive. After launching a CVM, execute the following command in a different terminal than the one with the login prompt:

```
$RV_ROOT_DIR/t1_test-primitive.sh
```

If it succeeds, a green checkmark followed by "Successfully..." will appear on the screen when the script finishes.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The frequency distributions of plaintexts in the CVM are preserved when encrypted at the same system physical address (sPA), which can be exploited to learn the ciphertexts corresponding to the prevalent values in the CVM, such as zero (Section 1, Section 3.1).

(C2): The sparsity information in the guest page tables can be exploited to significantly reduce the number of possible guest virtual addresses (GVA) of the `glibc` library (Section 4, exemplified on `nginx`).

(C3): The sparsity information in the OpenVDB node buffers can be used to leak the structural information of the objects being processed under OpenVDB library operations (Section 5).

(C4): The sparsity information in the ReLU buffers can be decoded and utilized to leak information about tokens by placing probes on the blurred activation information (Section 6).

A.4.2 Experiments

Claim C1 is supported by experiment **E1-C1**. For claim C2, the main supporting experiment is **E2-C2**, plus an optional experiment **E3-C2**. In contrast to the main experiments, which complete within minutes, the experiments marked as (optional) take significantly longer (several hours) and are included for completeness. We support claim C3 with the main experiment **E4-C3** followed by optional experiments **E5-C3**, **E6-C3**, and **E7-C3**. Finally, we support claim C4 with the experiment **E9-C4** preceded by the experiment **E8-C4**.

(E1-C1): [**learning-zero-ciphertexts**] [1 compute-minutes + 3 human minutes] In this experiment, the attacker learns the ciphertexts corresponding to plaintext zeros in the CVM on the target page.
Preparation: (1) Launch a victim CVM:

```
$RV_ROOT_DIR/launch-vm1.sh # VM1 as an example
```

Execution: Once the CVM is launched, in another window (or shell, log-in session)[1], execute

```
$RV_ROOT_DIR/e1_ciphertext-learning.sh
```

Results: If it succeeds, a green check followed by "Successfully..." will appear on the screen when the script finishes. A detailed description of the contents of collected logs, including `$RV_ROOT_DIR/cl.log` and `$RV_ROOT_DIR/dmesg-output.log` is in `README.md`.

(E2-C2): [**ASLR**] [Around 6 compute-minutes + 5 human-minutes]: This experiment is a minimal demonstration of the ASLR de-randomization scenario, exemplified by de-randomizing the `glibc`'s base address of `nginx` with a single memory layout.
Preparation: (1) Launch the victim **VM1** for ASLR.

```
$RV_ROOT_DIR/launch-vm1.sh
```

Execution: Execute

```
$RV_ROOT_DIR/e2_aslr.sh
```

Results: The output logs are located in `$RV_ROOT_DIR/aslr/aslr-min-log/`. The detailed structure of this directory is provided in the "Description of the Output Logs" section of the `README.md` file. Upon successful execution, the directory `0/nginx-worker/` should contain a file whose name ends with `window_size_64.log`. The results of non-windowed and windowed trackings can be checked at the end of `0/nginx-worker/windowed_evaluation.log`, starting with the line "ASR metrics for service nginx-worker".

(E3-C2): [**ASLR-full (optional)**][5-6 compute-hours + 10 human-minutes] The spirit of this experiment is the same as E2, but extends to all five applications in Section 4 and evaluates 10 layouts for each. As a result, it takes around 50× more time than it takes for the **E2-C2** experiment.
Preparation: The same as **E2-C2**.
Execution: Execute

```
$RV_ROOT_DIR/e3_aslr-full.sh
```

Results: The output logs will be in `$RV_ROOT_DIR/aslr/aslr-all-log/`. They are similar to **E2-C2** but with more memory layouts and more applications for each layout. Upon completion, execute the following scripts to parse the logs and produce data for Table 2 in the paper.

```
cd $RV_ROOT_DIR/aslr
python ./scripts/aslr_track_analysis.py
```

(E4-C3): [**openvdb-traversal**] [10 compute-minutes, 15-25 human-minutes] This experiment leaks the object's structural information from OpenVDB's read-only traversal operation.
Preparation: (1) Launch the victim CVM **VM2**.

```
$RV_ROOT_DIR/launch-vm2.sh
```

Execution: Execute

```
$RV_ROOT_DIR/e4_openvdb-traversal.sh
```

Results: The scripts produce logs in `$RV_ROOT_DIR/openvdb-leak/openvdb-traversal-1-time/`. A detailed description of its structure is in `README.md` of the artifact package. In this directory, the file `output_all.txt` contains the page fault events during tracking. The scripts process the record of page-fault events and output `PLY` files (a 3D data format commonly used to

[1] Depending on your preference, you can achieve multiple shells by using `tmux` or logging in to a new session.

store point clouds and polygonal meshes). When the recovery succeeds, it should contain eight PLY files in evaluation_traversal_extraction_<timestamp>_1/<ply-file-dir>/.

Visualizing the extracted pieces and assembling them. Synchronize the directory <ply-file-dir> and the Blender project file $RV_ROOT_DIR/openvdb-leak/blender_projects/traversal.blend to your local machine. The object labeled "original" in the Blender project corresponds to the source object shown in Figure 8(a), with the source file provided as $RV_ROOT_DIR/openvdb-leak/blender_projects/original.ply.

Import the extracted PLY files in <ply-file-dir> into Blender and manually reassemble the eight fragments by adjusting their positions along the X, Y, and Z axes, as illustrated in $RV_ROOT_DIR/openvdb-leak/blender_projects/blender-adjust-xyz.png. Once assembled, toggle visibility to show only the extracted pieces, and export them as a single file, traversal.ply. We also provide analysis tools in $RV_ROOT_DIR/openvdb-leak/blender_projects/. For example, visualize_traversal.py visualizes traversal.ply, reproducing the effect shown in Figure 8. p2p_dist_traversal.py generates part of the comparison in Figure 9 of the paper, contrasting the distribution of nearest-neighbor distances between objects extracted from the traversal operation and randomly populated points.

(E5-C3): [**openvdb-traversal-3-trackings (optional)**] [Triple the time required for E4 + around 6 GB disk space for storing the logs.] Everything remains the same as in **E4-C3**, except the experiment is repeated three times for evaluation.
Preparation: The same as **E4-C3**.
Execution: The same as **E4-C3**'s execution, but pass a different number as the number of times to run as a parameter. Execute

```
$RV_ROOT_DIR/e5_openvdb-traversal-3.sh
```

Results: The same as **E4-C3** but applied to more tracking instances.

(E6-C3): [**openvdb-construction** (optional)] [Around 3 compute-hours + 15-25 human-minutes + around 80GB disk space for storing logs] This experiment leaks the object's structural information from OpenVDB's construction operation.
Preparation: The same as **E4-C3**.
Execution: Execute

```
$RV_ROOT_DIR/e6_openvdb-construction.sh
```

Results: The same as **E4-C3**, but applied to a different victim's operation.

(E7-C3): [**openvdb-construction-3-trackings (optional)**] [Triple the time required for E6-C3 + around 240 GB disk space for storing the logs.] Everything remains the same as in **E6-C3**, except the experiment is repeated three times.
Preparation: The same as **E4-C3**.
Execution: Execute

```
$RV_ROOT_DIR/e7_openvdb-construction-3.sh
```

Results: The same as **E6-C3** but applied to more tracking instances.

(E8-C4): [**sparse-llm-data**] [5–6 compute-hours + 3 human-minutes + around 230GB disk.] This experiment produces data for **E9-C4**.
Execution: Execute

```
$RV_ROOT_DIR/e8_sparse-llm-data.sh
```

Results: The result directory $RV_ROOT_DIR/sparsellm-probe/powerinfer_reduced_activation_probes_results/ is generated.

(E9-C4): [**sparse-llm-figures**] [1 compute-minute + 3 human-minutes] Visualize the results of **E8-C4**.
Preparation: Finish **E8-C4**.
Execution: Execute

```
$RV_ROOT_DIR/e9_sparse-llm-figures.sh
```

Results: Figures 11 and 12 in the paper can be produced as probe_results.png and world_large_reduce4.png in the sparsellm-probe directory when the execution succeeds.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: GPUHammer: Rowhammer Attacks on GPU Memories are Practical

Chris S. Lin[†]
University of Toronto
shaopenglin@cs.toronto.edu

Joyce Qu[†]
University of Toronto
joyce.qu@mail.utoronto.ca

Gururaj Saileshwar
University of Toronto
gururaj@cs.toronto.edu

A Artifact Appendix

A.1 Abstract

This artifact supports the paper "GPUHammer: Rowhammer Attacks on GPU Memories are Practical", which presents the first successful Rowhammer attack on GPUs. The paper introduces GPUHammer, a framework that reverse-engineers GDDR6 DRAM row mappings, employs GPU-specific memory access patterns to intensify hammering, and bypasses existing mitigations. Using this approach, we demonstrate bit flips on an NVIDIA A6000 GPU and show how they can be exploited to tamper with machine learning models, leading to severe accuracy degradation. The artifact includes the GPUHammer implementation and evaluation scripts used to replicate (1) Rowhammer campaigns on A6000 GPU (Table-1), (2) Characterization of Rowhammer bit-flips in A6000's GDDR6 (Figure 11, Figure 12, Table 3), and (3) Exploits on ML models (Figure 13, Table 4).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact triggers Rowhammer faults on GPU memory and may induce bit-flips. Although these are contained within GPU DRAM and the scope of evaluation is limited, evaluators should not run this artifact on production systems or GPUs used for other sensitive tasks.

Ensure that ECC is disabled and GPU settings are configured as described. Use at your own risk.

A.2.2 How to access

The artifact is hosted on Zenodo and will become public once the embargo on this work lifts on August 12, 2025: Zenodo link: https://doi.org/10.5281/zenodo.15612689

A.2.3 Hardware dependencies

NVIDIA A6000 GPU with GDDR6 memory (48 GB)

[†]Equal contribution.

A.2.4 Software dependencies

- **Rowhammer Bit Flip Characterization**
  - Ubuntu 20.04+
  - NVIDIA CUDA Toolkit 12.3
  - NVIDIA CUDA Driver $\geq$ 545.23.08
  - g++ ($\geq$11.4.90, C++17)
  - CMake $\geq$ 3.26.4
  - Python $\geq$ 3.10
- **Exploit**
  - Anaconda 24.9.2 (Already installed on our machine)
  - RAPIDS RMM (Installed by artifact setup scripts)

A.2.5 Benchmarks

ImageNet 2012 Validation Set: **ILSVRC2012_img_val.tar**, required for the exploit on ML models, can be downloaded from https://www.image-net.org/download.php.

A.3 Set-up

A.3.1 Installation

Download Artifact and Datasets:

1. Please download the artifact from the Zenodo link provided in Section A.2.2. The downloaded file should be named `gpuhammer-main.zip`. Please unzip this folder.
2. Download the ImageNet validation set from appendix A.2.5 and move it inside the unzipped folder.

Setup environment:

```
cd gpuhammer-main
export HAMMER_ROOT=`pwd`

# Install Dependencies
bash ./run_setup.sh

# Enable rmm_dev environment and build GPUHammer
conda init
```

```
source activate base
conda activate rmm_dev
cmake -S $HAMMER_ROOT/src \
      -B $HAMMER_ROOT/src/out/build
cd $HAMMER_ROOT/src/out/build
make
cd $HAMMER_ROOT
```

Listing 1: Setup instructions

A.3.2 Basic Test

Generate a small Conflict-Set with the following command:

```
cd gpuhammer-main
python3 ./util/run_timing_task.py conf_set \
--threshold 27 \
--step 256 \
--it 15
```

Listing 2: Commands to generate a Conflict-Set

Expected output: The script should generate a file `CONF_SET.txt` that has 80 lines. The first 8 numbers expected in the file are: `852224, 852736, 868352, 868864, 895232, 895744, 911360, 911872`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): GPUHammer induces Rowhammer bit-flips on NVIDIA A6000 GPUs. We validate this with experiment E1 in Section 7.1, whose results are reported in Table 1.

(C2): The bit-flips on A6000 GPUs have the following characteristics: (1) Rowhammer Threshold (T_{RH}) of $\approx 12.3K$, TRR sampler size of 16, and bidirectional bit-flips. This is demonstrated by experiments E1 and E2 described in Sections 7.3-5, whose results are reported in Figure 11, Figure 12, and Table 3.

(C3): In a few memory massaging attempts, bit flips can be induced in ML model weights to significantly degrade model accuracy. This is demonstrated in experiment E3 described in Section 8.3, whose results are reported in Figure 13 and Table 4.

A.4.2 Experiments

The detailed steps to run the experiments are provided in the README.md file. In the following subsections, we provide an overview of the steps to run the experiments. The following commands automatically run all the experiments (E1, E2, E3) which will take a total of 3.5 days:

```
cd gpuhammer-main
bash ./run_artifact.sh
```

Listing 3: Commands to run all experiments

Below, we list the steps run by the script in listing 3.

(E0): ConflictRow-Set Generation (1 day): The following generates the Row-Set and Conflict-Set for the banks that will be hammered in subsequent experiments. If these have already been run, then you can skip this step.

```
cd gpuhammer-main
bash ./run_row_sets.sh
```

Listing 4: Commands to generate Conflict/Row-Sets

(E1) Systematic Hammering Campaigns: (1 day) This experiment performs 24-sided hammering patterns on the entire memory of the 4 different DRAM Banks (A, B, C, D) listed in the paper. On executing this experiment, the evaluators should see some (if not all) of the 8 flips in Table 1 (C1), and be able to observe the characteristics of those bit-flips in Table 3 (C2). One may see only a subset of the flips as due to time constraints we limit the attempts to trigger a bit flip on any given row.

Execution: To execute the 24-sided systematic hammering campaign on all four DRAM banks, run:

```
cd gpuhammer-main
bash run_t1_t3.sh
```

Listing 5: Execute systematic hammering campaign

Results: The results of the campaign in the folder `results/campaign/`, generate the last column of Table 1, and Table 3. The last column of Table 1 in `t1.txt`, shows the number of bit-flips in each of the 4 DRAM banks. Table 3 in `t3.txt` shows the information about each bit-flip, such as flip-direction, location, etc. We additionally provide the sample output of the Rowhammer campaign in `results/sample`, as the tables generated by the evaluator may not be complete due to the random nature of Rowhammer bit-flips.

(E2) Bit-flip Characterization (5 hours): We run two experiments that characterizes the Rowhammer Threshold (T_{RH}) and the TRR Sampler Size. On running these experiments, the key results to be reproduced include: (1) T_{RH} of the 8 bit-flips to be $12K$-$16K$ as shown in Figure 11 (C2), and that (2) bit-flips are only observed for 17-sided patterns and beyond as per Figure 12 (C2).

Execution: To execute these experiments, run:

```
cd gpuhammer-main
bash run_fig11.sh
bash run_fig12.sh
```

Listing 6: Execute bit-flip characterization experiments

Results: The results of the T_{RH} and TRR sampler size characterizations are in folders `results/fig11` and `results/fig12`, respectively, and the scripts recreate Figure 11 and Figure 12 stored as `fig11.pdf` and `fig12.pdf`.

(E3) ML Model Exploit (1.5 days): We craft exploits to degrade accuracy of ML models with the bit-flips. The experiments massage the GPU memory to force placement of model weights in victim locations followed by hammering to inject bit-flips in these victim locations. We perform this on 5 models with 50 massaging attempts each. The results to be reproduced include: (1) worst case model accuracy and Relative Accuracy Drop (RAD) with each bit-flip for all the models similar to Table 4 (C3), (2) Attack attempt vs RAD for bit-flip D1 (Figure 13), showing 99% RAD in < 10 attempts (C3).

Execution: To perform the exploit, run:

```
cd gpuhammer-main
bash run_fig13_t4.sh
```

Listing 7: Execute ML model exploit

Results: The results of T_{RH} and sampler size characterization are in the folder `results/fig13_t4`. Table 4 and Figure 13 are automatically generated from the result and stored as `t4.txt` and `fig13.pdf`, respectively.

A.5 Notes on Reusability

The README provides more details including:

- Re-configuration of the code to work on other GPUs.
- Extending the hammering by modifying the hammering parameters (e.g., aggressor pattern length, tuning the delays for synchronization).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: SCASE: Automated Secret Recovery via Side-Channel-Assisted Symbolic Execution

Daniel Weber
CISPA Helmholtz Center for Information Security

Lukas Gerlach
CISPA Helmholtz Center for Information Security

Leon Trampert
CISPA Helmholtz Center for Information Security

Youheng Lü
SCHUTZWERK GmbH

Jo Van Bulck
DistriNet, KU Leuven

Michael Schwarz
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This paper proposes SCASE, a novel methodology for inferring secrets from an opaque victim binary using symbolic execution guided by a concrete side-channel trace. Our key innovation is utilizing the memory accesses observed in the side-channel trace to prune the symbolic-execution space, thus avoiding state explosion. To demonstrate the effectiveness of our approach, we introduce Athena, a proof-of-concept framework to recover secrets from Intel SGX enclaves via controlled channels automatically. We show that Athena can automatically recover the 2048-bit secret key of an enclave running RSA and the 256-bit key from an RC4 KSA implementation. Furthermore, we demonstrate key recovery of OpenSSL's 256-bit AES S-Box implementation and recover the inputs to OpenSSL's binary extended Euclidean algorithm. To demonstrate the versatility of our approach beyond cryptographic applications, we further recover the input to a poker-hand evaluator. Our findings indicate that constraining symbolic execution via side-channel traces is an effective way to automate software-based side-channel attacks without requiring an in-depth understanding of the victim application.

A.2 Description & Requirements

The majority of our artifact only requires a Linux installation supporting Python 3 and a recent *angr*[1] installation. For recovering the 2048-bit RSA key, a machine with at least 64 GB of RAM is required. The only (optional) hardware requirement is required to regenerate the memory traces for the 2048-bit RSA key. For regenerating the memory traces for the 2048-bit RSA key, which we consider an optional experiment as we provide the traces alongside our artifact, an Intel SGX machine is required.

[1] https://github.com/angr/angr

A.2.1 Security, privacy, and ethical concerns

Our framework does not pose any security, privacy, or ethical concerns, as it only executes Python code. Nevertheless, the (not required) execution of our *ptrace tracer* requires the user to disable ASLR. For this, scripts exist to dis- and enable ASLR in the corresponding folder. Furthermore, our artifact contains real-world code used as meaningful victim applications. While the source code of these applications is publicly available, we cannot guarantee that the code is free of security, privacy, or ethical concerns. To the best of our knowledge, the code does not contain any such concerns.

A.2.2 How to access

The artifact is publically available on GitHub https://github.com/cispa/scase.
Furthermore, the artifact is archived on Zenodo with DOI https://doi.org/10.5281/zenodo.15609410. Note that, when fetched from Zenodo, the file `scase-traces-*.zip` needs to be extraced to `./traces`.

A.2.3 Hardware dependencies

Our artifact requires a Linux machine with at least 62 GB of allocatable RAM. The optional regeneration of the memory traces for the RSA key requires an Intel SGX machine.

A.2.4 Software dependencies

Our installation instructions are written for Ubuntu 22.04, but the artifact should work on most recent Linux distributions. We require Python 3.10 or higher, an *angr* installation, and at least 62 GB of allocatable RAM.

A.2.5 Benchmarks

Our artifact contains the memory traces used during the evaluation of our framework. These memory traces are CSV files

containing the memory accesses of the victim application. While we provide instructions on regenerating the memory traces, users are free to use the provided memory traces.

A.3 Set-up

A.3.1 Installation

After cloning the repository, create a virtual environment and install the required dependencies described under "Dependencies" in the `athena/README.md` file.

A.3.2 Basic Test

To run a basic test, execute the following command (after loading the virtual environment and from within the folder `./athena`): `python3 ./hex_elf.py`. Upon successful execution, the output should contain `Encoded solution: 8`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Athena can automatically recover secrets for varying granularities of memory traces, which are not recovered without memory trace guidance. This is shown by Experiment E1 (Section 5.1.1) and Experiment E6 (Sections 5.1.2 and 5.2).

(C2): Athena can recover an AES key from OpenSSL's S-Box implementation and the input of OpenSSL's binary extended Euclidean algorithm implementation. This is shown by Experiment E2 (Section 5.3) and Experiment E3 (Section 5.4).

(C3): Athena can recover the input to an RC4 KSA implementation. This is shown by Experiment E4 (Section 5.5).

(C4): Athena can recover the input to non-cryptographic applications, as demonstrated by its applicability to the TPT hand evaluator. This is shown by Experiment E5 (Section 5.6).

A.4.2 Experiments

To ease the reproduction of our results, we provide precompiled victim applications and memory traces in the `traces` folder. The Athena framework uses angr to perform symbolic execution on these binaries. Thus, the scripts, also referred to as *Athena wrappers* in the latter, never actually execute these binaries but rather emulate their execution. Experiment E7 (optional) describes how to recompile the victim applications and regenerate the memory traces. Note that the regeneration of the traces requires an Intel SGX machine with SGX-Step[2] installed, which is not required for the other experiments.

[2] https://github.com/jovanbulck/sgx-step

(E1): [Jump-Table Example] *[30 human-minutes + 2 compute-hours]:* This experiment shows Athena recovering the key from our Jump-table toy victim.
How to: The Athena wrapper for this can be found in `./athena/hex_elf_eval.py`. Our results, which are used in Figures 5 and 6 of the paper, can be found in `traces/jump-table/`. The victim program's code can be found in `./victim-programs/jump-table/`
Preparation: Traverse to the folder `./athena` and load the installation's venv.
Execution: Execute `python3 ./hex_elf_eval.py` and `python3 ./hex_elf_eval_noprune.py`. This generates subfolders with the following structure: `eval_data_g_<bytelength>_<option>`. When `<option>` equals `noprune`, the symbolic execution is not pruned by the side-channel trace.
Results: The resulting statistics are contained in the file `statistics.csv` for each subfolder. These (accumulated) numbers correspond to the paper's Figures 5 and 6. The most noteworthy result is that the `<...>_noprune` subfolders should indicate that the value could not be recovered. This can be confirmed by checking the column `incorrect_bytes` in the corresponding `statistics.csv` file.

(E2): [AES S-Box] *[30 human-minutes + 30 compute-minutes]:* This experiment shows Athena recovering the symmetric key from OpenSSL's AES S-Box implementation.
How to: The Athena wrapper for this can be found in `./athena/aes_openssl.py`. The victim program's code can be found in `./victim-programs/openssl-aes-sbox/`.
Preparation: Traverse to the folder `./athena` and load the installation's venv.
Execution: Execute `python3 ./aes_openssl.py`. This recovers the 256-bit AES key and prints whether the key is correct. The key can be verified by checking the file `./victim-programs/openssl-aes-sbox/victim.c`. The command `python3 ./aes_openssl_eval.py` can be used to execute the experiment for different memory trace granularities.
Results: The execution of `aes_openssl.py` should print that the key was successfully recovered. The execution of `aes_openssl_eval.py` creates two folders `eval_data_g01` and `eval_data_g01_dfonly`, where the latter restricts the guidance to the data-flow trace. Both folders contain a file `statistics.csv`, which contains the detailed statistics (similar to E1). The number of incorrect bytes for e should align with the paper's Table 1.

(E3): [BEEA] *[30 human-minutes + 1 compute-hour]:* This experiment shows Athena recovering the key from OpenSSL's binary extended Euclidean algorithm

(BEEA) implementation.

How to: The Athena wrapper for this can be found in `./athena/beea_openssl.py`. The victim program's code can be found in `./victim-programs/openssl-beea/`.

Preparation: Traverse to the folder `./athena` and load the installation's venv.

Execution: The experiment can be executed by running `python3 ./beea_openssl.py`. This recovers the second argument to `BN_gcd`. Note that this takes around 10-14 hours to finish.

Results: The execution of `aes_openssl.py` should print the key contained in the variable `p_min_one` from `./victim-programs/openssl-beea/main.c`.

(E4): [RC4-KSA] *[30 human-minutes + 5 compute-minutes]:* This experiment shows Athena can recover the RC4 key from a key scheduling algorithm (KSA) implementation.

How to: The Athena wrapper for this can be found in `./athena/rc4_elf.py`. The victim program's code can be found in `./victim-programs/rc4-ksa/`.

Preparation: Traverse to the folder `./athena` and load the installation's venv.

Execution: Execute `python3 ./rc4_elf.py`. This recovers the key from `victim-programs/rc4-ksa/main.c`.

Results: The execution of `rc4_elf.py` should print the key recovered from the binary. This key should match with the content of `./victim-programs/rc4-ksa/key.hex`.

(E5): [TPT Hand Evaluator] *[30 human-minutes + 2 compute-minutes]:* This experiment shows Athena can recover secrets from non-cryptographic applications, in this case, the input to a poker-hand evaluator.

How to: The Athena wrapper for this can be found in `./athena/poker_elf.py`. The victim program's code can be found in `./victim-programs/tpt-hand-evaluator/`.

Preparation: Traverse to the folder `./athena` and load the installation's venv. Additionally, unpack the binaries contained in the tar balls in `./traces/tpt-hand-evaluator/`. For this, run `find . -name '*.tar.gz' -exec tar -xzvf {} \;` from within the folder.

Execution: The experiment can be executed by running `python3 ./poker_elf.py`. This recovers the card array from `victim-programs/tpt-hand-evaluator/test.c`.

Results: The execution of `poker_elf.py` should print the array contained in the variable `cards` in `./victim-programs/tpt-hand-evaluator/test.c`. As we executed this program multiple times with varying inputs, the folder `./traces/tpt-hand-evaluator/` does not only contain the binaries and memory traces but also the expected outcomes for each input. The combinations of input and expected output are enumerated. For example, the files `cftrace1.csv`, `dftrace1.csv`, `victim1`, and `solution1.txt` belong together.

(E6): [Intel SGX Square+Multiply Enclave] *[2 human-hours + 10 compute-minutes]:* This experiment recovers a 2048-bit key from an Intel SGX enclave implementation of a toy Square+Multiply algorithm.

How to: The Athena wrapper for this can be found in `./athena/sm_enclave.py`. The victim program's code can be found in `./victim-programs/square-multiply-enclave/`.

Preparation: Traverse to the folder `./athena` and load the virtual environment from the installation.

Execution: The experiment can be executed by running `python3 ./sm_enclave.py`.

Results: The printed key should match the variable `secret` contained in the file `square-multiply-enclave/encl.c`.

(E7 - Optional): [Victim Recompilation and Trace Regeneration] *[4 human-hours + 2 compute-hours]:* This experiment is about recompiling the victim applications and regenerating the memory traces provided in `./traces/`.

How to: For each victim application, the corresponding folder contains a `README.md` file with complication instructions, e.g., the instructions to recompile the Jump-Table example are found in `./victim-programs/jump-table/README.md`.

Preparation: The Jump-Table example does not require any special preparation besides a `gcc` installation. The AES and BEEA examples require the corresponding OpenSSL library to be compiled. The hand evaluator requires a `g++` installation. The Square+Multiply Enclave requires Intel SGX and SGX-Step installed.

Execution: Follow the instructions in `victim-programs/<victim>/README.md` to recompile the victim application and regenerate the memory traces.

Results: The binaries and traces should match the ones provided in `./traces/<victim>`. Note that due to varying environments, it is unlikely that the binaries and resulting traces are identical. One can copy the resulting binaries and memory traces to the corresponding folder in `./traces/` rerun the corresponding experiment.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Shadows in Cipher Spaces: Exploiting Tweak Repetition in Hardware Memory Encryption

Wei Peng[*] Yinshuai Li[*] Yinqian Zhang[†]
Research Institute of Trustworthy Autonomous Systems
Department of Computer Science and Engineering
Southern University of Science and Technology

A Artifact Appendix

A.1 Abstract

Our paper introduces the CIPHERSHADOW attack, which exploits the reuse of the same tweak value in the 64-byte memory blocks of the Hygon CSV v1, v2, v3 memory encryption mechanisms. This reuse leads to successful decryption through 16-byte ciphertext replacement, and the characteristic that identical 16-byte ciphertexts have the same plaintext. CIPHERSHADOW can tamper with the program's control flow and data flow by substituting encrypted memory after interrupting the VM at an attacker-chosen point in the program execution. This artifact includes two attacking scenarios: a malicious SSH login implemented to CSV v1 and v2, and a one-shot ciphertext leakage of the MNIST dataset implemented to CSV v1, v2, and v3.

A.2 Description & Requirements

This artifact comprises four main experiments: 1) Gadget scanning; 2) Proof of Concept (PoC) of SSH-malicious login; 3) Page location; 4. PoC of one-shot ciphertext leakage. The experiments must be conducted on Hygon processors. For the SSH-malicious login and page location, the processor must be capable of enabling VMs with CSV v1 or v2. For the one-shot ciphertext leakage, the processor needs to be able to enable VMs with CSV v1, v2, or v3.

A.2.1 Security, privacy, and ethical concerns

Observing the ciphertext does not pose a security issue, but replacing 16-byte aligned ciphertext blocks may cause the VM to crash if the content of the program replacement is not correctly chosen or the timing of the replacement is incorrect.

A.2.2 How to Access

This artifact are publicly available on Zenodo: https://doi.org/10.5281/zenodo.15614377 and GitHub: https://github.com/pw0rld/CSV-CipherShadow.git.

A.2.3 Hardware dependencies

This attack must be executed on a Hygon-series CPU with at least CSV v1 support. Our experiments were conducted on the following processors:

- **Hygon C86 5285 CPU with CSV v1 support.**
- **Hygon C86 7390 CPU with CSV v2 support.**

A.2.4 Software Dependencies

Our experiments rely on the following software that supports enabling CSV VMs:

- **QEMU** (`csv-devel-6.2.0-v1`)[*]
- **Linux Kernel** (`OLK-6.6`)[†]
- **OVMF** (`csv-stable202202-v1`)[‡]

A.3 Set-up

After installing and building the compatible versions of QEMU, the kernel, and OVMF, you can use the following commands to start CSV.

```
git clone github.com:pw0rld/CSV-CipherShadow
cd CSV-CipherShadow
./make_vm_img.sh
sudo ./start-qemu.sh
```

[*]These authors contributed equally to this work.
[†]Corresponding Author: yinqianz@acm.org

[*]https://gitee.com/anolis/hygon-qemu/tree/csv-devel-6.2.0-v1/
[†]https://gitee.com/openeuler/kernel/tree/OLK-6.6/
[‡]https://gitee.com/anolis/hygon-edk2/tree/csv-stable202202-v1/

A.3.1 Basic Test

We provide a simple memory observation experiment to demonstrate the existence of this vulnerability. If the vulnerability exists, you should observe repeated 16-byte aligned ciphertext patterns within 64-byte blocks for identical plaintexts.

```
cd Observation_memory && make
insmod ob.ko
dmesg
```

A.4 Evaluation workflow

The experiments we have prepared include application fingerprinting, SSH malicious login, and a one-shot ciphertext leakage attack.

A.4.1 Major Claims

(C1): *We implemented a gadget scanner (Section 5) to find gadgets in target programs that can manipulate program control flow and data flow, by replacing aligned ciphertexts of 16 bytes within 64 bytes.*

(C2): *We demonstrate that a program fingerprint can be constructed based on the pattern of whether the 16-byte ciphertexts within a 64-byte block are identical, which helps in locating program pages (Section 6.1).*

(C3): *We demonstrate that* CIPHERSHADOW *hijacks program control flow and tampers with function return values by replacing 16-byte ciphertexts, enabling malicious login to SSH services (Section 6.2).*

(C4): *We demonstrate that in the CSV memory encryption mechanism, the characteristic of 16-byte aligned identical ciphertexts within 64 bytes having the same plaintext can lead to one-shot ciphertext leakage attacks (Section 6.3).*

A.4.2 Experiments

(E1): Gadget Scanning:

How to: Select the sensitive functions in the application you want to attack and search for gadgets to help tamper with the control flow or data flow, thereby achieving the purpose of tampering with the return value.

Preparation: Download the necessary Python dependencies.

```
pip install -r requirements.txt
```

Execution: Execute the gadget scanner script, the first parameter is the target program's binary file, the second parameter specifies the function for finding gadgets, and the third parameter configures specific values of registers and memory at the function entry.

```
python gadget_search.py <Target Binary>
    <Target Functions> <Initial State
    Value>
```

Results: You can see the results of the gadget scan in the terminal, including the number of illegal and potential control flow and data flow gadgets, along with the corresponding assembly content of each gadget.

(E2): SSH-Malicious Login [10 minutes]:

How to: The attacker is able to log into the TEE VM using an incorrect password through the SSH service. In this experiment, we demonstrate that the encryption mechanism of CSV allows for replacing 16 bytes within a 64-byte block and successfully decrypting, and by utilizing the ciphertext pattern to locate pages, we hijack the return value of `sys_auth_passwd` function by substituting the code for failed login attempts with that of successful ones.

Preparation: First, the attacker needs to preprocess the SSH binary to identify the characteristic features of the page containing the *sys_auth_passwd* function. Then, using page faults, the attacker obtains suspicious VM pages. After filtering a specific set of VM pages, the attacker extracts their memory features and compares them with the preprocessed features. A match indicates that the page is the target page. If multiple matches occur, further methods are required for precise identification, such as leveraging the register page features discussed later.

Notice: If the machine freezes when starting CSV, you can run the reload.sh script in the repository. This script will reload the kernel and remove previous cached configurations.

Execution: The tool automatically targets the currently running CSV VM, repeatedly attempts SSH logins until the SSH login interface is identified, and modifies the login logic to validate successful authentication.

```
sudo python3 analysis_csv.py
```

Results: The attacker is able to log into the server using an incorrect password.

(E3): Page location [10 minutes]:

How to: In this experiment, we use a benchmark consisting of a set of assembly instructions with special patterns. We extract their characteristics using single-stepping. This allows us to observe the execution of instructions and locate the target page.

Preparation: First, the attacker needs to compute the register characteristics of a specific assembly code segment within the target page. This process requires manual analysis using GDB. In this experiment, we use a 14-bit value to record the change patterns of different registers. If a register changes compared to the previous state, the corresponding bit is set to 1. In this experiment,

the register feature we use is: [0x1, 0x80, 0x40, 0x10, 0x8, 0x4, 0x4, 0x8, 0x10, 0x40, 0x80, 0x1].

Execution: This command triggers a timer interrupt and scans a designated Guest Physical Address (GPA) to extract its physical page characteristics. The provided parameters include the physical address and the timer interval.

```
./poc single_step_page 0x10af9e000 10
dmesg -c > dmesg.log
```

Noticed: Please note that due to differences in machine APICs, testers may need to identify a suitable APIC Timer value on their own machines. On our machines, the effective range is between 7 and 10.

Results: The attacker can observe the register behavior of the page through its GPA, and leverage this feature to locate the page by matching it with the preprocessed binary.

(E4): One-Shot Ciphertext Leakage [5 minutes]:

How to: For CSV v3, read and write protection has been implemented on encrypted memory, preventing attackers from performing ciphertext replacement attacks. However, we can infer and recover the training set for the MLP-Classifier model by utilizing the characteristic that identical 16-byte aligned plaintexts have identical ciphertexts within a 64-byte block through ciphertext patterns.

Preparation: Since we do not have physical access to the CSV v3 machine, we simulated the one-shot memory collection process in CSV v2. After the training set was loaded into memory, we completed the dump of the encrypted memory. The file `leak.bin` contains the collected training set memory data.

Execution: Run the leakage attack script, which includes three steps: recovering images, training the model, and validating the success rate of recovering images.

```
python leakage-attack.py
```

Results: The restored images will be in the `of_dir` folder, and the terminal will output the success rate of the image recovery.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Websites' Global Privacy Control Compliance at Scale and over Time

Katherine Hausladen[1], Ebuka Akubilo[1], Matthew May[1], Francisca Wijaya[1], Jocelyn Wang[2], Oliver Wang[1], Sophie Eng[1], and Sebastian Zimmeck[1]

[1]*Department of Mathematics and Computer Science, Wesleyan University*
[2]*Department of Computer Science, Princeton University*

A Artifact Appendix

A.1 Abstract

The California Consumer Privacy Act (CCPA) and its regulations give California residents the right to opt out of the sale or sharing of their personal information. Similar privacy rights exist in other jurisdictions as well. These rights can be exercised via Global Privacy Control (GPC) [9]. GPC is a binary privacy preference signal that can be implemented in browsers, browser extensions, mobile operating systems, and other software. GPC signals are sent via HTTP headers or can be set via a JavaScript DOM property. When a first-party site receives a GPC signal, it must propagate users' opt out preferences to integrated third parties, such as ad networks. This propagation is done using privacy strings, such as the Global Privacy Platform (GPP) String [5]. To evaluate websites' compliance with the CCPA opt out right expressed via GPC, we implemented our main artifact: a GPC web crawler based on the Selenium WebDriver [8] and a browser extension for analyzing privacy string propagation. We based our browser extension on the OptMeowt extension in analysis mode [6], as described and implemented by Zimmeck et al. [10]. Our browser extension identifies sites' GPC compliance by sending GPC signals to sites and detecting the presence of privacy strings, their values, and value changes in response to receiving GPC signals. In addition to our GPC web crawler we also provide a Python script for decoding GPP Strings, crawl data, and privacy policy analysis results, the latter of which we used to identify selling and sharing of personal information from first to third parties.

A.2 Description & Requirements

The artifact requires a consumer-grade computer with Docker [3] and Python [7] (version 3.10 or later) installed.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

All artifacts are available in our Zenodo repository [4].

A.2.3 Hardware dependencies

While there are no particular hardware dependencies, we tested the functionality described here using a MacBook with macOS operating system.

A.2.4 Software dependencies

We use Docker [3] for the GPC web crawler and Python [7] for the GPP privacy string decoding.

A.2.5 Benchmarks

None

A.3 Set-up

1. Create a Docker account [2].
2. Authenticate to Docker Hub [1].
3. Install Python [7].

A.3.1 Installation

- Download and unzip *gpc-web-crawler.zip* from the Zenodo repository.
- Download and unzip *cmp_api_python.zip* from the Zenodo repository.

A.3.2 Basic Test

GPC Web Crawler Test

1. Open a new terminal session.
2. Navigate to the *gpc-web-crawler-main* directory. This directory was created by unzipping *gpc-web-crawler.zip*.

3. From within that directory run `make custom`.

4. Once completed, the results from the crawl are saved to the *./crawl_results* directory. The directory the results are saved to is prefixed with "CUSTOMCRAWL" and suffixed with the timestamp you started the crawl with. Verify that there is an entry for `yelp.com` in each of the *analysis.json* and *well-known-data.csv* files.

Python Installation Check

1. Open your terminal.

2. Run the following command:

```
python3 --version
```

3. If the command returns a version earlier than 3.10 or indicates Python is not installed, download and install Python 3.10 or later from the official website: https://www.python.org/downloads/.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *We identify sites' compliance with the CCPA opt out right when receiving GPC signals at scale and over time.*

A.4.2 Experiments

(E1): *[~10 compute-minutes]: The main experiment is running the GPC web crawler on a batch of sites. The result of the experiment is a dataset of 10 sites' privacy string behavior when receiving GPC signals.*
Preparation: It is assumed that you have followed the set-up instructions (§A.3).
Execution: Run the experiment as follows:

1. Open a new terminal session.
2. Navigate to the *gpc-web-crawler-main* directory that you created when unzipping *gpc-web-crawler.zip*.
3. Ensure no Docker containers are running by executing the command `make stop`.
4. Run `make start`.
5. When prompted to select a batch (0-8), select 0.
6. Once completed, the result are saved to the *./crawl_results* directory. The directory the results are saved to is prefixed with "CRAWLSET0" and suffixed with the timestamp you started the crawl with.

Results: Verify the contents of CRAWLSET0. *analysis.json* and *well-known-data.csv* should contain the privacy string analysis results for 10 sites. An error-logging subdirectory will also be created, recording extra data collected during crawl failures. In that subdirectory, there will be a file named *error-logging.json*, a JSON file associating domains with the exception that caused the crawler to fail. If possible, a screenshot of the site will also be taken and saved in this subdirectory.

(E2): *[~5 developer minutes]: The second experiment consists of decoding GPP Strings.*
Preparation: It is assumed that you have followed the set-up instructions (§A.3).
Execution: Run the experiment as follows:

1. Open a new terminal session.
2. Extract the file `cmp_api_python.zip`.
3. Navigate into the resulting `cmp_api_python` directory:

```
cd cmp_api_python
```

4. Run the decoding script with Python:

```
python3 decode.py
```

5. When prompted, enter the GPP string `DBABBg~BUUAAAGA.YA`.

Results: Verify the output matches the following object:

```
{"uscav1": {
        "Version": 1,
        "SaleOptOutNotice": 1,
        "SharingOptOutNotice": 1,
        "SensitiveDataLimitUseNotice": 0,
        "SaleOptOut": 1,
        "SharingOptOut": 1,
        "SensitiveDataProcessing": [0, 0,
0, 0, 0, 0, 0, 0, 0, 0],
        "KnownChildSensitiveDataConsents":
[0, 0],
        "PersonalDataConsents": 1,
        "MspaCoveredTransaction": 2,
        "MspaOptOutOptionMode": 0,
        "MspaServiceProviderMode": 0,
        "GpcSegmentType": 1,
        "Gpc": True,
    }
}
```

A.5 Notes on Reusability

You can use your own set of sites to crawl with the GPC web crawler and GPP string to decode with the GPP String decoder.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] Docker Inc. Authenticate to Docker Hub with web-based login. https://docs.docker.com/reference/cli/docker/login/#authenticate-to-docker-hub-with-web-based-login. Accessed: April 3, 2025.

[2] Docker Inc. Create your account. https://app.docker.com/signup. Accessed: April 3, 2025.

[3] Docker Inc. Install Docker Engine. https://docs.docker.com/engine/install/. Accessed: April 3, 2025.

[4] K. Hausladen, E. Akubilo, M. May, F. Wijaya, J. Wang, O. Wang, S. Eng, and S. Zimmeck. GPC Web Crawler. https://doi.org/10.5281/zenodo.14729170. Accessed: April 3, 2025.

[5] IAB Tech Lab. Global-Privacy-Platform. https://github.com/InteractiveAdvertisingBureau/Global-Privacy-Platform. Accessed: April 3, 2025.

[6] privacy-tech-lab. OptMeowt. https://github.com/privacy-tech-lab/gpc-optmeowt/tree/v3.0.0-paper. Accessed: April 3, 2025.

[7] Python Software Foundation. Python. https://www.python.org/. Accessed: April 3, 2025.

[8] Software Freedom Conservancy. WebDriver. https://www.selenium.dev/documentation/webdriver/. Accessed: April 3, 2025.

[9] S. Zimmeck, P. Snyder, J. Brookman, and A. Zucker-Scharff. Global Privacy Control (GPC). https://w3c.github.io/gpc/. Accessed: April 3, 2025.

[10] S. Zimmeck, O. Wang, K. Alicki, J. Wang, and S. Eng. Usability and Enforceability of Global Privacy Control. In *23rd Privacy Enhancing Technologies Symposium (PETS 2023)*, pages 265–281, Lausanne, Switzerland, July 2023. PETS.



USENIX Security '25 Artifact Appendix: Atkscopes: Multiresolution Adversarial Perturbation as a Unified Attack on Perceptual Hashing and Beyond

Yushu Zhang[1], Yuanyuan Sun[1], Shuren Qi[1,*], Zhongyun Hua[2], Wenying Wen[3], Yuming Fang[3]

[1]Nanjing University of Aeronautics and Astronautics
[2]Harbin Institute of Technology, Shenzhen
[3]Jiangxi University of Finance and Economics
[*]Corresponding author: Shuren Qi, Email: shurenqi@nuaa.edu.cn

A Artifact Appendix

A.1 Abstract

For USENIX Security '25 Artifact Evaluation, we provide the code, models, datasets, and execution scripts of our paper. In addition, we provide a detailed description in this document on how to set up the environment and reproduce the main statements of our paper.

A.2 Description & Requirements

The attack against PhotoDNA has been tested on a 64-bit Windows machine. This limitation is due to the PhotoDNA binary, which is architecture-specific. The attacks on phash, PDQ, and NeuralHash have been tested on Linux. It is important to note that the attack on NeuralHash requires a CUDA-supported GPU.

To reproduce our major results, you can run our attack as described in the README.md.

A.2.1 Security, privacy, and ethical concerns

This artifact is used to perform adversarial attacks on perceptual hashing algorithms. Although these attacks are conducted in a controlled and compliant manner in the paper, evaluators should avoid applying these attacks to real-world systems or platforms, especially without authorization, as this may violate terms of service or cause unintended damage to commercial or personal systems.

A.2.2 How to access

Please download the artifact zip file from Zenodo . After extracting the file, please open the two subdirectories below as separate projects in PyCharm, as these projects need to be built in different environments.

A.2.3 Hardware dependencies

The only dependency is the need for a machine compatible with the corresponding .dll or .so file. This should work in a 64-bit Windows environment.

A.2.4 Software dependencies

The main dependencies are Docker, TensorFlow, and PyTorch. Other dependencies are listed in the requirements.txt.

A.2.5 Benchmarks

In the artifact, we provide executable Python files for escaping and triggering regulation attacks on pHash, PDQ, PhotoDNA, and NeuralHash. We evaluate our two attack scenarios using the ImageNet dataset from the ILSVRC 2012 challenge. For our experiments, we have randomly selected 50 pairs of images from the ImageNet dataset, which are available in the directories `./imagenet_50_resized/` and `./imagenet_tar_50_resized/`. Additionally, we have included the necessary models for the experiments in the artifact. The model for computing visual loss can be found in the `./lpips/` directory. The models for pHash, PDQ, PhotoDNA, and NeuralHash used in the attacks are located in the following files: `./imagehash.py`, `./python/pdqhashing`, `./PhotoDNAx64.dll`, and `./NeuralHash` respectively.

A.3 Set-up

A.3.1 Installation

To conduct attacks on shallow hashes (pHash, PDQ, PhotoDNA) and deephash (NeuralHash), two different environments need to be configured. Below, we will first explain how to configure the environment to run the Python files under `AtkScope_ShallowHash`. It is important to note that the PhotoDNA model can only run on Windows systems. Therefore,

if you wish to conduct attacks on PhotoDNA, you will need to set up the environment on a Windows system.

First, you need to download the Anaconda installation package from the official website. Once the installation is complete, open the terminal and enter the following commands:

```
$ conda create -n <your_env_name> python=3.6

$ conda activate <your_env_name>

$ pip install -r requirements.txt
```

For running files under `AtkScope_NeuralHash`, we recommend configuring the environment on a Linux system. Please run the following command from the project's root

```
$ sudo docker build -t hashing_attacks --build-arg
     USER_ID=$(id -u) --build-arg GROUP_ID=$(id -g
    ) .

$ sudo docker build -t hashing_attacks -f rootless
    .Dockerfile .
```

A.3.2 Basic Test

Run the following command in the project root directory to test if the shallow hashing environment is working properly.

```
$ conda activate <your_env_name>

$ python test_attack_rgb_target_PDQ.py  --
    untargeted -a black -d imagenet -c 10 -o 10000
     -m 10000  --reset_adam -n 50 --solver adam -b
     1 -p 1 --hash 92 --use_resize --htype "PDQ"
    --init_size 32 --init_dct_size 16 --method "
    tanh" --modifier_method "multiply" --batch 16
    --gpu 1 --lr 1 -s "
    RGB_results_imagenet_targetPDQ_lpips_dct8_lr0
    .1_c10_dist92" --start_idx=0 --dist_metrics "
    pdist" --save_ckpts "
    best_modifier_imagenet_target_PDQ"
```

The expected correct output is shown in Figure 1.

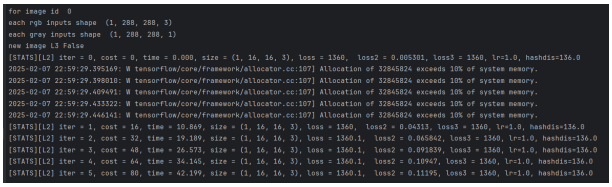


Figure 1: Expected output of testing the shallow hashing environment.

Run the following command in the project root directory to test if the deep hashing environment is working properly.
To start the docker container run the following command from the project's root:

```
$ sudo docker run --rm --shm-size 16G --name
    my_hashing_attacks --gpus '"device=0"' -v $(
    pwd):/code -it hashing_attacks bash
```

To run the triggering regulation attack against NeuralHash, enter the following command in the project root :

```
$ python adv1_target_attack.py --source=
    imagenet_50_resized --target_hashset=
    dataset_hashes/imagenet_tar_50_resized_hashes.
    csv --output_folder '
    output_target_imagenet_l2_dist17_each10_lr0.01
    '
```

The expected correct output is shown in Figure 2.

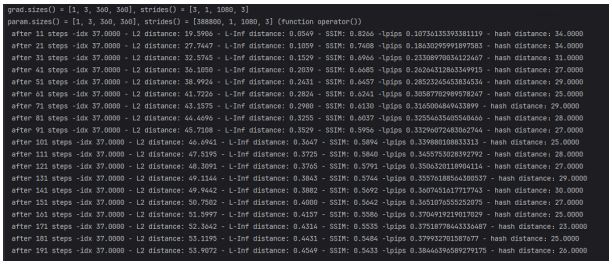


Figure 2: Expected output of testing the deep hashing environment.

A.4 Evaluation workflow

We will include the operational steps and experiments that must be performed to evaluate if our artifact is functional and to validate our paper's key results and claims in this section.

A.4.1 Major Claims

(C1): Atkscopes achieve highly efficient and effective adversarial attacks on four commercial hashing algorithms—pHash, PDQ, PhotoDNA, and NeuralHash, in both the escaping regulation and triggering regulation attack scenarios. This is proven by the experiments (E1) and described in sections 5.3, with results reported in tables 2 and 3.

A.4.2 Experiments

(E1): [Escaping and Triggering Regulation Attacks] [10 human-minutes + 32 compute-hour + 20GB disk]:
Preparation: Follow the configuration instructions for the two environments as described in section A.3. For the PhotoDNA attack experiment, the setup must be done on a Windows system. All instructions can be found in the README file, and we recommend copying the commands directly from the README to avoid formatting errors.

Execution: Open the README file, copy the attack commands into the terminal in the project root directory to run the Python scripts.
Results: For the Python files under `AtkScope_ShallowHash`, after the program finishes running, the terminal prints `success_rate`, which corresponds to the Success Rate in the table, `overall average l2`, which corresponds to the L_2 distance in the table, `overall average perceptual distance`, which corresponds to the LPIPS in the table, `overall average iterations`, which corresponds to the Rounds in the table, and `overall average hash_ori` and `overall average hash_tar`, which correspond to the HashDistance in the table.
For the Python files under `AtkScope_NeuralHash`, after the program finishes running, the terminal prints `Success rate`, which corresponds to the *Success Rate* in the table, `Average L2 distance`, which corresponds to the L_2 *Distance* in the table, `Average LPIPS distance`, which corresponds to the *LPIPS Distance* in the table, `Average iterations`, which corresponds to the *Rounds* in the table, and `Average target loss`, which corresponds to the *HashDistance* in the table.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: How Transparent is Usable Privacy and Security Research? A Meta-Study on Current Research Transparency Practices

Jan H. Klemmer [C] Juliane Schmüser [C] Fabian Fischer [C] Jacques Suray [C]
Jan-Ulrich Holtgrave [C] Simon Lenau [C] Byron M. Lowens [*] Florian Schaub [†]
Sascha Fahl [C]

[C]CISPA Helmholtz Center for Information Security, Germany
[*]Indiana University Indianapolis, USA
[†]University of Michigan, USA

A Artifact Appendix

A.1 Abstract

To support open science and transparency, we provide several artifacts that accompany our paper. The artifacts contain the data used for our analysis (including the raw analysis results, as well as computed data like the transparency score), analysis scripts to compute the transparency score, and results such as tables and figures for the paper, and the R code for the regression, as well as the generated outputs. Moreover, we include the PDFs that detail the results of our sample size estimation.

Please refer to the contained `README.md` for a more detailed overview of which artifacts are contained in which folders and files.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Except for installing dependencies, the software does not receive/send Internet data. All computations are done locally. The code does not execute any destructive steps. There are no security, privacy, or ethical concerns or risks for the evaluators.

A.2.2 How to access

The artifacts are hosted at Zenodo and accessible via the following link: https://doi.org/10.5281/zenodo.15532982.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

The contained source code can be executed on any system that fulfills the following requirements:

- Python with Poetry as dependency manager (Python dependencies are listed in `code/pyproject.toml`). Poetry can be downloaded here: https://python-poetry.org/.
- Docker (Installation: https://docs.docker.com/engine/install/).

A.2.5 Benchmarks

Our dataset with analysis results is contained in the artifacts and does not need to be installed separately.

A.3 Set-up

Please confirm that the required software packages are installed on the system as stated above (i.e., Python, Poetry, and Docker), before proceeding with the following steps.

A.3.1 Installation

1. Visit https://doi.org/10.5281/zenodo.15532982 and download the artifacts. Under the tab "Files" you can choose to download the `.zip` file.
2. Unarchive the `.zip` file on your local computer.
3. Run the following commands to install Python dependencies: `cd code/` and `poetry install`
4. For the regression that we implemented in R, we are using a Docker container. To build the docker container,

execute the following commands: `cd code/R/` and then `./build-container.sh`.

A.3.2 Basic Test

- For the Python scripts: Open a shell (from the `code/` directory) in the virtual environment with `poetry shell`. Then execute `jupyter notebook`. This should open Jupyter in your browser. (If that is not the case, you can manually navigate your browser to `http://localhost:8888/`). Try to open one of the notebooks (`.ipynb`).
- For the Docker container: `docker run -it --rm containr:latest /bin/bash` (opens a shell).[1]

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Transparency Criteria Availability:* For 52 transparency criteria, we quantify the transparency in terms of availability. On average, 65.0% ± 11.2 of the papers' applicable criteria are available (cf. Section 6.1); in total, 63.9% of all criteria were available, 5.2% partially available, and 30.9% unavailable. We present the detailed availability for all 52 transparency criteria in Table 3.

(C2): *Transparency Score:* To assess the overall transparency of papers, we calculate a transparency score (TS). The TS per paper is on average 0.677 ± 0.103 (cf. Section 6.3.2).

(C3): *Regression:* The exploratory regression on the TS reveals a significantly positive association of transparency with a paper's length and fewer methods. Most venues do not differ significantly from the baseline (SOUPS). The model indicates no significant relation with AE, a paper's primary method, and publication year. (Exact results contained in Table 4 and Section 6.4.2.)

A.4.2 Experiments

(E1): *[Transparency Analysis] [15 human-minutes + 0 compute-hour + <1GB disk]:* Process the result dataset obtained from the manual paper annotation and calculate availabilities, transparency scores, etc.
Preparation: Open a shell (from the `code/` directory) in the virtual environment with `poetry shell`. Then execute `jupyter notebook`. This should open Jupyter in your browser. (If that is not the case, you can manually try to navigate your browser to `http://localhost:8888/`).
Execution: First, open `transparency-score.ipynb` in Jupyter and execute all cells (*"Run"*, *"Run All Cells"*). This will transform the raw annotation results stored in `transparency-analysis.tsv`, calculate the transparency score and store all results in `results.tsv`. Next, open `main.ipynb` in Jupyter and execute all cells (*"Run"*, *"Run All Cells"*). Based on `results.tsv`, this notebook contains the actual analysis, e.g., to generate some of the figures and tables.
Results: The overall availability (C1) is presented in cells 19 and 20 of `main.ipynb`, and the table written to `output/tables/criteria.tex`. The results of the transparency score (C2) are written to `data/result.tsv` (column: "Transparency Score"), and the descriptive statistics are in cell 10.
Apart from that, all computed numbers are also stored (for usage in LaTeX) as key-value-pairs in `numbers-generated.tex`.

(E2): *[Regression] [15 human-minutes + 30 compute-minutes + 6GB disk]:* Run the regression model (and related other R code).
Preparation: Build the docker container as described above.
Execution: Run `docker run -it --rm -v $(pwd)/data:/data containr:latest /bin/bash` (from the `code/` directory) to open a shell in the docker container and mount your local `data/` directory. Once you entered the container environment, run `./main` to start the analysis.
Results: You can now retrieve all results in the `data/` folder on your local machine (or inside the container at `/data`). This also includes the regression table (C3).

A.5 Notes on Reusability

Our artifacts operate on the contained dataset. Therefore, the code is reusable on other datasets containing transparency analysis data (assuming that they are in the same format as our dataset). For example, future work might analyze the same transparency criteria, store the transparency analysis data in a similar `.tsv`/`.csv` file, and run our code to compute transparency scores.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2025/`.

[1] **For Mac/Apple Silicon** it might be necessary to setup emulation first.
Option A): Enable Emulation in Docker Desktop (should be enabled by default): docker.desktop settings → general → virtual machine options. Choose *Apple Virtualization framework* and activate *Rosetta*.
Option B): Set Platform Explicitly When Pulling or Running: If Docker does not find an ARM version of an image, it may fail unless you explicitly request the `amd64` version: `docker run --platform linux/amd64 <image>`



USENIX Security '25 Artifact Appendix: "Helps me Take the Post With a Grain of Salt:" Soft Moderation Effects on Accuracy Perceptions and Sharing Intentions of Inauthentic Political Content on X

Filipo Sharevski
DePaul University

Verena Distler
Aalto University

Florian Alt
Ludwig-Maximilians-Universität München
University of the Bundeswehr Munich

A Artifact Appendix

A.1 Abstract

The artifact contains two datasets and associated code that analyzes the data for two studies. We performed two studies: **study 1** during the US election 2024 and a replication **study 2** post the elections. The code runs a linear regression model with the soft moderation interventions (warnings/covers in the order of friction) as *independent variable* while the *dependent variable* is: *perceived accuracy* (**RQ1**); *sharing intentions* (**RQ2**). The code also runs a linear regression model with the same variables only controlling for *intention to vote* (**RQ3**) and *demographics*. The data used for the thematic analysis (**RQ5**) is also included in column I titled `sharedqual` and column J titled `accuracyqual` for the sharing intentions and the accuracy perceptions qualitative responses, respectively.

A.2 Description & Requirements

To recreate our analysis for each dataset, RStudio is needed. All the files needed for the functional testing and reproducibility verification is included in the zip archive `manip_media_warnings_replication_files`. The archive contains a `README.md` that describes its structure, consisting of two folders: `study1` and `study2` each including subfolders for `code` and for the *data*, respectively. The `code` subfolders contain an `analyze.qmd` script that runs in RStudio, set up to perform the full quantitative analysis (**RQ1** – **RQ4**) as described in the abstract above (throughout the analysis, we used the customary *p*-value threshold of 0.05, though we included additional thresholds of 0.1 and 0.01 for completeness and transparency). The `data` subfolders contain the `data_clean.Rds` data corresponding to the quantitative values used in the linear regression models, and `data_clean.xlsx` including the qualitative answers used as an input in the thematic analysis (**RQ5**). The `clean` qualifier indicates datasets with removed low quality responses as described in our data collection protocol. Each folder also contains a `tables` subfolder populated with the LaTeX table code export we used for reporting of the results in the paper.

A.2.1 Security, privacy, and ethical concerns

There are no risks for evaluators. The data collection was anonymous and we removed any entries that could potentially lead to identifying any of the participants in both `datasets`. The `code` only implements our linear regression models.

A.2.2 How to access

We are making the dataset as a research artifact available and also provide the scripts we used to analyze the data to allow for both functionality check and reproducibility. The study stimuli are publicly available: https://osf.io/pj895/files/osfstorage?view_only=76a18f0b13bb4281874b0e8d5b0b9bdf. Alternatively, the artifacts can be accessed at: https://doi.org/10.17605/OSF.IO/PJ895

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

No specific OS requirements. The software needed for running the linear regression analysis is Rstudio (https://posit.co/download/rstudio-desktop/). Microsoft Excel or any spreadsheet software could be used to open the datasets too, for the purpose of the thematic analysis.

A.2.5 Benchmarks

None.

A.3 Set-up

The analysis for each dataset is executed by running the file `analyze.qmd` in Rstudio. The following package are needed for running the analysis: "qualtRics", "tidyverse", "writexl", "readxl", "ggplot2", "broom", "performance", "see", "patchwork", "lmtest", "car", "estimatr", "sandwich", "stargazer".

You may need to add this line of code in the `analyze.qmd` script if it doesn't load them for some reason:

```
packages <- c("qualtRics", "tidyverse",
"writexl", "readxl", "ggplot2",
"broom", "performance", "see",
"patchwork", "lmtest",
"car", "estimatr",
"sandwich", "stargazer")
install.packages(packages)
```

A.3.1 Installation

Download the replication files from the link above.

A.3.2 Basic Test

The replication files are structured first by study (**study 1** and **study 2**). Within each study, you will find the following folders:

- `code/`: Contains analysis script (`analyze.qmd`).
- `data/`: Contains data from each dataset.
- `tables/`: Contain the LaTeX tables generated by the analysis scripts. Run the code to produce the tables.

Run the analysis script (`analyze.qmd`)

A.4 Evaluation workflow

The replication files are structured first by study (**study 1** and **study 2**). Within each study, you will find the following folders:

- `code/`: Contains analysis script (`analyze.qmd`).
- `data/`: Contains data from each dataset.
- `tables/`: Contain the LaTeX tables generated by the analysis scripts. Run the code to produce the tables.

Run the analysis script (`analyze.qmd`) for (**RQ1–RQ4**). For **RQ5** the qualitative responses are in column I titled `sharedqual` and column J titled `accuracyqual` for the sharing intentions and the accuracy perceptions in the `data_clean.xlsx` for each study. The results for **RQ5** are inherently subjective per the Braun & Clarke's *thematic analysis* methodology.

A.4.1 Major Claims

(C1): RQ1: in both studies, the warning covers (high level friction) and the community notes bundled with warning labels (medium level friction) had a statistically significant negative effect on the perceived accuracy

(C2): RQ2: in both studies, none of the soft moderation interventions had a statistically significant effect on the sharing intentions;

(C3): RQ3: for both studies, we included an interaction term for the soft moderation interventions and the *intention to vote* to check whether the soft moderation's effect varied depending on the voting intentions. The interaction term was not statistically significant

(C4): RQ4: in both studies, frequent content sharing on X was associated with a higher perceived accuracy and of inauthentic content

(C5): RQ4: in study 1, participants who had voted in previous US election cycles showed lower perceived accuracy for the all the stimuli tested compared to participants who had never voted before for a US president

(C6): RQ4: in study 2, younger participants also showed lower perceived accuracy for all the study stimuli tested.

(C7): RQ4: in study 1, participants living in suburbs near a large city reported lower sharing intentions compared to participants living in a large city

(C8): RQ4: in study 2, participants living in rural areas reported lower sharing intentions compared to participants living in a large city.

(C9): RQ4: in both studies, frequent content sharing on X was associated with a higher sharing intentions of inauthentic content

(C10): RQ5: in both studies, the perceptions of accuracy depend on the perceived trustworthiness of the surrounding context. Compared to community notes, labels, and third-party fact-checks were viewed as less credible. A politically incongruent personal disposition toward the subject of the content tended to shift perceived accuracy toward authenticity, even when the content was demonstrably inauthentic.

(C11): RQ5: Sharing inauthentic (soft moderated) content on social media is restrained by the perceived risk of reputation and relationship damage: Sharing might happen as an act of responsibility, though, counting on the warning labels and community notes as a counterbalance against legitimizing misinformation on social media.

A.4.2 Experiments

A short introduction to interpreting linear regression tables (based on Backhaus et al., 2021): In the regression table, the **intercept** represents the expected value of the dependent variable when all independent variables are zero. In multivariate linear analysis, The **coefficients** denote the change in the dependent variable per unit of change in the independent variable, all other independent variables held constant. A positive coefficient indicates a positive relationship and a negative coefficient indicates a negative relationship. Coefficient **standard errors** are in brackets. The standard error of the coefficient is an indicator of the precision (lower standard

errors mean higher precision of estimates). We used the customary statistical **significance level** of 0.05. R-squared values reflect the proportion of variance of the dependent variable explained by the independent variables. In the social sciences, relatively lower R-squared values are often expected because the concepts studied are necessarily influenced by many factors outside of the experimental conditions. The **adjusted R-squared value** accounts for the number of parameters and the sample size, meaning that it penalizes increasing model complexity or overfitting. For more details, see Backhaus et al. (2021), chapter 2.

For each study, once in the `study1` or `study2` folder:

(E1): RQ1 [10 human-seconds + 20 compute-seconds + 0.5MB disk]:
How to: Open the analysis script (`analyze.qmd`) in RStudio
Preparation: `analyze.qmd` is self-contained. No preparation needed.
Execution: `Run all` when the the analysis script (`analyze.qmd`) in RStudio
Results: The results are populated in `model_1_accuracy_warning_table.tex`.
Interpretation: The warning covers and the community notes bundled with warning labels as a medium level friction had a statistically significant negative effect on the perceived accuracy.

(E2): RQ2 [10 human-seconds + 20 compute-seconds + 0.5MB disk]:
How to: Open the analysis script (`analyze.qmd`) in RStudio
Preparation: `analyze.qmd` is self-contained. No preparation needed.
Execution: `Run all` when the the analysis script (`analyze.qmd`) in RStudio
Results: The results are populated in `model_2_share_warning.tex`.
Interpretation: None of the soft moderation interventions had a statistically significant effect on the dependent variable sharing intentions.

(E3): RQ3 [10 human-seconds + 20 compute-seconds + 0.5MB disk]:
How to: Open the analysis script (`analyze.qmd`) in RStudio
Preparation: `analyze.qmd` is self-contained. No preparation needed.
Execution: `Run all` when the the analysis script (`analyze.qmd`) in RStudio
Results: The results are populated in `model_3a_3b_table.tex`.
Interpretation: We included an interaction term for the frictions and the intention to vote to check whether the soft moderation's effect varied depending on the voting intentions. The interaction term was not statistically significant, for any level of soft moderation intervention friction. Warning labels bundled with community notes or warning covers decreased the perceived accuracy of inauthentic content on X.

(E4–E9): RQ4 [10 human-seconds + 20 compute-seconds + 0.5MB disk]:
How to: Open the analysis script (`analyze.qmd`) in RStudio
Preparation: `analyze.qmd` is self-contained. No preparation needed.
Execution: `Run all` when the the analysis script (`analyze.qmd`) in RStudio
Results: The results are populated in `model_4a_4b_table.tex`.
Interpretation: We included an interaction term for the frictions and the intention to vote to check whether the soft moderation intervention varied across the voting intentions. The interaction was not statistically significant. The frictions did not have a statistically significant effect on sharing intention.

(E10): RQ5 [30 human-hours + 5 compute-seconds + 0.5MB disk]:
How to: Open the excel spreadsheet (`data_clean.xlsx`) in Microsoft Excel
Preparation: Locate the column J titled `accuracyqual`
Execution: This is not *execution* per se, but this requires an repetition of the thematic analysis process as described by Braun & Clarke
Results: The results are any resultant themes developed through the thematic analysis process.

(E11): RQ5 [30 human-hours + 5 compute-seconds + 0.5MB disk]:
How to: Open the excel spreadsheet (`data_clean.xlsx`) in Microsoft Excel
Preparation: Locate the column I titled `sharedqual`
Execution: This is not *execution* per se, but this requires an repetition of the thematic analysis process as described by Braun & Clarke
Results: The results are any resultant themes developed through the thematic analysis process.

A.5 Notes on Reusability

The quantitative data in both studies could be reused to run other statistical analyses, given that is rich with variables of many sorts. The data used in the thematic analysis for both studies could be well reused for performing other types of thematic analyses; no adaptation is needed; the data might contain grammatical or syntactical mistakes given it is provided *as is*, at the time of collection.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Robust, Efficient, and Widely Available Greybox Fuzzing for COTS Binaries with System Call Pattern Feedback

Jifan Xiao[1], Peng Jiang[2], Zixi Zhao[1], Ruizhe Huang[1], Junlin Liu[1], and Ding Li[1]

[1] *Key Laboratory of High Confidence Software Technologies, Peking University*
[2] *Southeast University*

A Artifact Appendix

A.1 Abstract

This document provides instructions to execute the artifacts. This is a prelinimary version of SPFuzz and SPFuzz++. The artifacts are provided as a Virtual Machine (VM) image. The VM is configured with all the dependencies and the source codes of SPFuzz and SPFuzz++, with necessary data for benchmarks. Please exactly follow the instructions in this document to execute SPFuzz and SPFuzz++.

A.2 Description & Requirements

To use the artifacts, you first need to create a VM with the provided harddisk image. The image is in vdi format and can be loaded using Oracle VirtualBox. The VM is configured with 2 cores and 4GB of RAM. The VM is running Ubuntu 18.04 LTS. (Since the current version of NoDrop module does not support docker environments, we choose to share our artifacts in this way.)

A.2.1 Security, privacy, and ethical concerns

We have reported all the vulnerabilities discovered by SPFuzz to the authors/owners of the vulnerable software.

A.2.2 How to access

We publish the artifacts through the following link: https://doi.org/10.5281/zenodo.15209966.

Once you have configured and created the VM successfully, you can login to the VM using the following credentials:

Username: usenix
Password: security

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

For this version, the software environment should be identical with the provided VM image, especially the OS kernel version.

A.2.5 Benchmarks

Benchmarks are contained in the VM image disk.

A.3 Set-up

A.3.1 Installation

- Step 1: Configure Core Pattern For Fuzzers

```
sudo -s
echo core >/proc/sys/kernel/core_pattern
su usenix
```

- Step 2: Load NoDrop Kernel Module

```
cd ~/SPFuzz/NoDrop/build
bash remake.sh
make
make load
```

- Step 3: Make SPFuzz & SPFuzz++

  Take SPFuzz++ as an example (SPFuzz is similar) :

```
cd ~/SPFuzz/SPFuzz++
make clean
make
```

  Then, an executable named *afl-fuzz* will be generated in the current directory.

- Step 4: Prepare Benchmarks

  All benchmarks are held in the `~/SPFuzz/benchmarks` directory. Each benchmark is a directory with a binary file, an *input.zip* file containing the inputs, and another

.zip file containing the source codes if this benchmark is open-source.

To prepare the benchmark, you need to unzip the seeds from the *inputs.zip* file, and copy the binary to a new binary named *toTest* in the same directory. For example, to prepare the *perlbench_r* benchmark:

```
cd ~/SPFuzz/benchmarks/perlbench_r
unzip inputs.zip
cp perlbench_r toTest
```

A.3.2 Basic Test

Take SPFuzz++ as an example (SPFuzz is similar). First, make sure the "NoDrop" kernel module is loaded. Then, execute the following command:

```
cd ~/SPFuzz/SPFuzz++
AFL_SKIP_BIN_CHECK=1 ./afl-fuzz \
-t 1000 -m none \
-i ~/SPFuzz/benchmarks/NAME/inputs \
-o ~/SPFuzz/benchmarks/NAME/output \
-- ~/SPFuzz/benchmarks/NAME/toTest @@
```

Note to replace the NAME with the benchmark you want to test. An example using *perlbench_r* is stored in the `~/example.sh` file.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *SPFuzz and SPFuzz++ are able to fuzz all the benchmark binaries without source codes, and the efficiency is comparable with traditional methods.*

A.4.2 Experiments

(E1): *[Optional Name] [1 human-hour + 24 X 29 compute-hour + 3GB X 29 disk]:*
How to: First, prepare the tools and benchmarks according to Section A.3, then start the fuzzing campaign and wait for 24 hours. This procedure applies to all the 29 benchmarks, both open-source and close-source.
Results: Follow the instructions in step-6 of the README.md file contained in the repo to calculate coverage. SPFuzz and SPFuzz++ should be able to run all the fuzzing tasks. SPFuzz++ should obtain comparable speeds and coverage data with traditional fuzzers.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: BLuEMan: A Stateful Simulation-based Fuzzing Framework for Open-Source RTOS Bluetooth Low Energy Protocol Stacks

Wei-Che Kao
National Yang Ming Chiao Tung University

Yen-Chia Chen
National Yang Ming Chiao Tung University

Yu-Sheng Lin
National Yang Ming Chiao Tung University

Yu-Cheng Yang
National Yang Ming Chiao Tung University

Chi-Yu Li
National Yang Ming Chiao Tung University

Chun-Ying Huang
National Yang Ming Chiao Tung University

A Artifact Appendix

A.1 Abstract

Bluetooth Low Energy (BLE) is widely used for low-power, short-range communication but remains vulnerable in some implementations. Existing fuzzing methods often lack scalability and platform flexibility. To address this, we present BLuEMan, a simulation-based fuzzing framework combining a Real-Time Operating System (RTOS) with a software-based physical layer simulator. BLuEMan runs actual BLE stacks and emulates device interactions, enabling faster, scalable testing across platforms. It achieves fuzzing speeds up to 18x and 162.3x faster than existing simulation- and platform-based methods, respectively. It has uncovered four new CVE-assigned vulnerabilities, offering an efficient path for BLE security testing.

This artifact provides the source codes, scripts, and instructions to reproduce the primary evaluation results present in the paper, namely packet sending rate, fuzzing coverage, packet selection strategy, and vulnerability discovery.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact poses no security or privacy risks to artifact reviewers, testing runtime, or their users. The tools run everything in a container and can be run in parallel on multiple instances when the resources are affordable.

A.2.2 How to access

The artifacts are publicly accessible at the following platforms, including Zenodo, GitHub, and DockerHub.

- Zenodo: https://doi.org/10.5281/zenodo.15601101
- GitHub: https://github.com/zoolab-org/blueman.artifact
- DockerHub: https://hub.docker.com/r/zoolab/blueman

A.2.3 Hardware dependencies

Evaluating the artifact requires a machine equipped with an Intel x86-64 compatible CPU. It is because BLuEMan depends on BabbleSim, which currently supports only x86-64 CPU architecture. Our experiments use a server equipped with an Intel(R) Xeon(R) Gold 5118 CPU and 48 GB RAM, running Debian 12 Linux OS. To minimize the cost of reproducing the results, we omit the experiments that require special embedded hardware devices or depends on other papers' artifacts.

A.2.4 Software dependencies

We pack all the required software dependencies into a single docker-based container. Please install a compatible docker runtime from either the official Docker website or the docker package that comes with a Linux distribution.

A.2.5 Benchmarks

None. No data is required by the experiments with the artifact reported in our paper.

A.3 Set-up

A.3.1 Installation

- Step 1. Clone the repository from GitHub or download the release files from zenodo.

```
git clone --recursive
↪ https://github.com/zoolab-org/blueman.artifact.git
```

- Step 2. Switch to the `./blueman.artifact` folder and ensure that the scripts have correct executable permission.

```
chmod +x ./build.sh ./run.sh
```

- Step 3. Build the container image that is required to run the experiments.

```
./build.sh
```

The script creates a container image named `blueman_demo`. It may take 10–60 minutes to build the image, depending on the hardware performance and network bandwidth. Alternatively, users may download our pre-built image from the DockerHub. The commands to pull the image and apply a correct tag for running the scripts are:

```
docker pull zoolab/blueman
docker tag zoolab/blueman blueman_demo
```

A.3.2 Basic Test

Once the container image is ready, use the '`run.sh`' script to run the experiments. The script comes with a few options to choose different runtime preferences, as shown in Figure 1. We merge multiple available options into a single line to save the spaces in the document.

```
Usage: ./run.sh <action> <mutator> [packet selection strategies]
↪ <execution duration> <output_dir>

Available actions:
  - gatt_write_peripheral, gatt_write_central
  - hr_peripheral, hr_central,
  - sm_pairing_peripheral, sm_pairing_central
  - le_credit_server, le_credit_client
  - ots_peripheral, otc_central

Available mutators:
  - field, afl, random

Available packet selection strategies (for field mutator only):
  - FIXED_PROB_10, FIXED_PROB_25, FIXED_PROB_50, FIXED_PROB_75,
  ↪ FIXED_PROB_100, SELECTIVE_25_75, SELECTIVE_75_25
  - RANDOM_PROB, MIXED_PROB

Available execution duration (in minutes):
  - 1, 10, 60, 120, 360, 720, 1440, 43200
```

Figure 1: The usage of the `run.sh` script.

Note that the `run.sh` script preserves outputs in the folder specified in the last argument. Please ensure that the folder exists in the filesystem. If multiple tests are run in parallel, please remember to use a different directory for each test.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): BLuEMan runs much faster than other existing approaches. This is proven by the experiment (E1), where we show that BLuEMan achieves a higher packet sending rate, as presented in Section 6.1. Note that the numbers for SweynTooth and BTFuzz are reported from the corresponding papers.

(C2): The field-aware mutation approach achieves higher coverage than random mutation and AFL-only mutators. This is proven by the experiment (E2), where we show the reported coverage for each evaluated approach. Due to the nature of randomness involved in the fuzzer implementations, the reported coverage numbers may differ for each invocation. However, the trends should be aligned with the results presented in Section 6.3.

(C3): The packet selection strategy shows diverse performance in terms of coverage exploration. This is proven by the experiment (E3), where we show the reported coverage for each evaluated packet selection strategy. Due to the nature of randomness involved in the fuzzer implementations, the reported coverage numbers may differ for each invocation. The results should be similar to those presented in Section 6.4.

(C4): BLuEMan has the capability to uncover new bugs (CVE-2024-3332) in the Zephyr BLE protocol stack. This is proven by the experiment (E4). Because all the CVEs are reported from different versions of Zephyr, we choose the fastest one for the reproduction process.

A.4.2 Experiments

(E1): *[Packet Rate] [30 human-minute + 1 compute-hour + 20GB disk]: Run BLuEMan and observe the packet sending rates.*

How to: Run the fuzzing process and check the status message output from the tool.

Preparation: Follow the steps described in Section A.3 to create the required docker image.

Execution: Run BLuEMan using the command:

```
mkdir /tmp/output-folder
./run.sh ots_peripheral field FIXED_PROB_50 10
↪ "/tmp/output-folder"
```

Please ensure that `/tmp/output-folder` can be mounted into the docker instance and is writable by the instance.

Results: A sample output message is shown below. The `total_elapse` field shows the elapsed time (in microseconds) for running the fuzzer, and the `total_packet_count` shows how many packets have been sent during the fuzzing process.

```
start_time: 1748974762595148, cur_time:
↪ 1748974825155642, total_elapse: 62560494,
↪ round_elapse: 2676575
round 28, exec_counts 280, current corpus:
↪ ./output/seed//id_00003248_0000000000000065_000045,
↪ attack_error_count 0, crash_count 0,
↪ timeout_count: 0, coverage: 3593, max coverage:
↪ 4223, queue size: 88, total_packet_count: 26540
```

(E2): *[Mutation Approach] [1 human-hour + n×(1∼24) compute-hour, where n is the number of traces collected for plotting the results + 60GB disk]: Run BLuEMan and measure the coverage for different mutation approaches.*

How to: Run the fuzzing process and collect the results from the corresponding output folders. Please note that in our paper, each plotted curve represents three 24-hour runs. Users may run the experiments for a shorter time to facilitate the reproduction process and then check the output results.

Preparation: Follow the steps described in Section A.3 to create the required docker image.

Execution: We recommend to run BLuEMan using the simplified command:

```
./auto_gen_rq3.sh <action> <duration>
```

The command invokes BLuEMan, uses `<action>` as the interacting app for generating testing corpora, and runs the testing for `<duration>` minutes. The available `<action>` and `<duration>` are exactly the same as those listed by the '`run.sh`' script. Note that the '`./auto_gen_rq3.sh`' command utilizes 4 CPU cores in parallel. Fuzzing is a computationally intensive process, and execution speed may vary depending on available computing resources. Users may selectively run commands as specified in the `EVALUATION_RQ3.md` document to suit their environment better.

Results: An example output from a one-minute test of the `sm_pairing_central` app is shown below.

```
$ ./auto_gen_rq3.sh sm_pairing_central 1
==> Running: sm_pairing_central field
==> Running: sm_pairing_central afl
==> Running: sm_pairing_central random
==> Generating coverage plot for sm_pairing_central
Written /root/plot/csv/afl.csv
Written /root/plot/csv/field.csv
Written /root/plot/csv/random.csv
Reading field.csv
Reading random.csv
Reading afl.csv
Coverage plot saved
==> Coverage plot generation completed for
↪ gatt_write_peripheral, duration time: 1 minutes.
↪ The plot is saved in
↪ sm_pairing_central_1_rq3/eval_sm_pairing_central/cov/
```

The generated PDF file is stored in the `sm_pairing_central_rq3/eval_sm_pairing_central/cov/` directory on the host.

(E3): *[Packet Selection Strategy] [1 human-hour + n×(1∼2) compute-hour, where n is the number of traces collected for plotting the results + 60GB disk]: Run BLuEMan and measure the coverage for different packet selection strategies.*

How to: Run the fuzzing process and collect the results from the corresponding output folders. Please note that in our paper, each plotted curve represents 2 hours of testing. Users may run the experiments for a shorter time to facilitate the reproduction process and then check the output results.

Preparation: Follow the steps described in Section A.3 to create the required docker image.

Execution: We recommend to run BLuEMan using the simplified command:

```
./auto_gen_rq4.sh <action> <duration>
```

The command behaves similar to '`./auto_gen_rq3.sh`'. Similarly, users may selectively run commands as specified in the `EVALUATION_RQ4.md` document to suit their environment better. Note that the '`./auto_gen_rq4.sh`' command utilizes 10 CPU cores in parallel.

Results: An example output from a one-minute test of the `ots_peripheral` app is shown below.

```
$ ./auto_gen_rq4.sh ots_peripheral 1
==> Running: ots_peripheral FIXED_PROB_10
... (omitted for saving spaces)
==> Generating coverage plot for ots_peripheral
Written /root/plot/csv/field_FIXED_PROB_10.csv
... (omitted for saving spaces)
Reading field_FIXED_PROB_10.csv
... (omitted for saving spaces)
Coverage plot saved
==> Coverage plot generation completed for
↪ ots_peripheral, , duration time: 1 minutes. The
↪ plot is saved in
↪ ots_peripheral_1_rq4/eval_ots_peripheral/cov/
```

The generated PDF file is stored in the `ots_peripheral_rq4/eval_ots_peripheral/cov/` directory on the host.

(E4): *[Vulnerability Discovery] [30 human-minute + 0.5 compute-hour + 20GB disk]: Run BLuEMan and wait for reported crashes due to triggered vulnerabilities.*

How to: Run the fuzzing process and observe the outputs from the console logs and the corresponding output folders.

Preparation: Follow the steps described in Section A.3 to create the required docker image.

Execution: Run BLuEMan using the command:

```
mkdir /tmp/output-folder
./run.sh gatt_write_central field FIXED_PROB_50 60
↪ /tmp/output-folder
```

The command uses `gatt_write_central` as the interacting app for generating testing corpora. Note that the CVE-2024-3332 vulnerability can only be triggered using the app.

Results: An example output from a 60-minute test of the `gatt_write_central` app is shown below.

```
$ mkdir /tmp/output-folder
$ ./run.sh gatt_write_central field FIXED_PROB_50 60
↪ /tmp/output-folder
...
start_time: 1748962190058918, cur_time:
↪ 1748962489441826, total_elapse: 299382908,
↪ round_elapse: 17087627
round 21, exec_counts 210, current corpus:
↪ ./output/seed//id_00001858_0000000000000105_000070,
↪ attack_error_count 5, crash_count 1,
↪ timeout_count: 58, coverage: 3040, max coverage:
↪ 4531, queue size: 105, total_packet_count: 17253
```

When a crash is detected, the total number of detected crashes is shown in the `crash_count` field.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space

Chuyang Chen
The Ohio State University
chen.13875@osu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

A Artifact Appendix

This artifact appendix is a self-contained document that describes the roadmap for the evaluation of the artifacts of the paper "ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space." It includes the hardware, software, and configuration requirements for the evaluation, as well as the process for reproducing the results and claims presented in the paper.

The artifacts have acquired the "artifact available" and "artifact functional" badges after the artifact evaluation. The full replication experiments for the "artifact reproduced" badge were not finished within the one-month time span of the artifact evaluation process due to the requirements of a significant amount of time and substantial computational resources. However, our testing shows that the artifacts function well to reproduce all the results in the paper with enough time and computational resources provided. Furthermore, we will continue to improve the artifacts for better usability and bug fixes after the artifact evaluation.

A.1 Abstract

The artifacts contain the implementation of ELFuzz, the source code of the benchmarks and baselines compared with ELFuzz in the evaluation, the evaluation results, and the data required to reproduce the evaluation. It also includes a Docker image that preserves the environment and setups of the evaluation to facilitate one-touch replication.

A.2 Description & Requirements

This section outlines the hardware and software requirements necessary to recreate the experiment environment and setups described in the paper, as well as the steps to acquire the benchmarks and baselines used in the evaluation.

A.2.1 Security, privacy, and ethical concerns

The artifact does not perform any operations that may compromise the security or privacy of the evaluators who execute it.

A.2.2 How to access

The replication package, containing the code and data, is published on Zenodo. The download link of the version at the time of artifact evaluation is https://doi.org/10.5281/zenodo.16741080. Updated versions will be hosted at https://doi.org/10.5281/zenodo.15833146. The source code of ELFuzz and the replication package is developed in the GitHub repository at https://github.com/OSUSecLab/elfuzz.

A.2.3 Hardware dependencies

The experiments require a GPU with 26 GiB VRAM and CUDA support to run a local CodeLlama-13B model. The intermediate data and results produced or downloaded by the experiments will occupy approximately 100 GiB of disk space. In our evaluation, we use 30 processes to accelerate the experiments. It is recommended that the evaluators' machines be equipped with CPUs with at least this number of cores. Fewer CPU cores do not affect the results of the experiments, but a longer time to finish the experiments should be expected. The specific GPU and CPUs used in the original evaluation are one NVIDIA H100 Tensor Core GPU and two AMD EPYC 7251 CPUs.

A.2.4 Software dependencies

The experiments do not depend on a specific operating system. A Docker installation that is compatible with version 24.0.7 is required. The host machine should also have the CUDA toolkit installed.

A.2.5 Benchmarks

Benchmarks used in the evaluation will be automatically cloned from their official Git repositories or downloaded from the official sites during the experiments. Dockerfiles from the Fuzzbench project[1] and the OSS-Fuzz project[2] are modified for replicable building environments for the six benchmarks other than `cvc5`. The Dockerfile for `cvc5` is implemented from scratch. All Dockerfiles are included in the source code tarball of the artifacts.

[1] https://github.com/google/fuzzbench
[2] https://github.com/google/oss-fuzz

A.3 Setup

This section describes how to set up and configure the environment to replicate the evaluation. It also includes instructions on the functionality validation of the replication package.

A.3.1 Installation

First, follow the official instructions[3] to install Docker. Then, set up the core pattern in the host machine as required by AFL++ later. Hereafter, "$" indicates user inputs, and ">" indicates program outputs.

```
$ echo core > /proc/sys/kernel/core_pattern
> ...
```

Now, import the Docker image and launch the container:

```
$ zstd -d elfuzz_docker_<timetag>.tar.zst
$ docker load --input elfuzz_docker_<timetag>.tar
$ mkdir -p /tmp/host
$ docker run --storage-opt size=100G \
             --cpus 30 \
             --add-host=host.docker.internal:host-gateway \
             -v /tmp/host:/tmp/host \
             -v /var/run/docker.sock:/var/run/docker.sock \
             --name elfuzz \
             -it ghcr.io/osuseclab/elfuzz:25.07.2
```

The commands should lead you into the container where the experiments will happen.

A.3.2 Initilizing the environment

After entering the Docker container, run the following commands to enable sibling containers.

```
$ sudo chown -R appuser:appuser /tmp/host/
$ elfuzz setup
> ...
$ exit
```

The two commands will exit the container. Now, restart the container for the settings to take effect.

```
$ start -ai elfuzz
```

Then, the following command will download all the data files from Zenodo and place them in the correct locations.

```
$ elfuzz download
```

You also need to configure your Hugging Face token:

```
$ elfuzz config --set tgi.huggingface_token <YOUR_TOKEN>
```

A.3.3 Functionality Validation

After initializing the environment, you can run mini-versions of the experiments presented in the paper to validate that all of them function well. These commands adopt the same settings as the original experiments, but decrease the parameters such as time limits and the number of evolution iterations for quick validation. Each command can take 5 minutes to 2 hours to complete. §A.4 will give detailed explanations of these commands.

```
$ elfuzz synth -T fuzzer.elfuzz \
               --use-small-model \
               --evolution-iterations 3 \
               jsoncpp
$ elfuzz synth -T grammar.glade re2
$ elfuzz synth -T semantics.islearn cvc5
$ elfuzz produce --time 10 -T glade jsoncpp
$ elfuzz minimize -T glade jsoncpp
$ elfuzz run rq1.seed_cov -T glade jsoncpp
$ elfuzz run rq1.afl --fuzzers glade \
                     --repeat 1 \
                     --time 60 \
                     jsoncpp
$ elfuzz run rq2.afl --fuzzers glade \
                     --repeat 1 \
                     --time 60 \
                     jsoncpp
$ elfuzz run rq2.triage
$ elfuzz run rq2.real_world --time 60
$ elfuzz run rq3
```

These commands should output messages like "ELFuzz fuzzers successfully synthesized" without being interrupted by errors.

[3] https://docs.docker.com/engine/install/

A.4 Evaluation Workflow

This section outlines the major claims presented in the evaluation part of the paper (§A.4.1). It describes how to conduct the experiments to fully reproduce the results presented in the paper, thereby supporting these claims (§A.4.2). However, these experiments require a significant amount of time and substantial computational resources. Thus, another option is to run the smaller-scale experiments (§A.4.2), which share the same settings as the full-scale experiments but are conducted for a shorter time or with fewer repetitions. Results produced by the small-scale experiments should also support our claims, although not 100% replications of those in our original paper. We will explain how to conduct them and draw the same claims from the results.

A.4.1 Major claims

The major claims of the paper are as follows:

- **C1.** Fuzzers synthesized by ELFUZZ significantly outperform the state-of-the-art generation-based fuzzers, respecting the coverage of the produced test cases and the promotion that the test cases bring to later mutation-based fuzzing processes. This is proven by the input generation experiments (E1) and the mutation-based fuzzing experiment (E2) in RQ1.
- **C2.** Fuzzers synthesized by ELFUZZ significantly outperform the state-of-the-art generation-based fuzzers when being used to find artificially injected bugs, and they can find previously unknown bugs in real-world SUTs. This is proved by the bug-finding experiments on bug-injected benchmarks (E3) and the real-world bug-finding experiment on `cvc5` (E4).
- **C3.** Fuzzer space contributes the most to the performance of ELFUZZ among all the components. This is proved by the input generation experiments (E5) and the LLM-driven evolution processes (E6).
- **C4.** Fuzzers synthesized by ELFUZZ are interpretable and extensible. This is proved by case studies on the fuzzer code (E7) and manual adaptation of ZEST, another input-generation technique, onto the fuzzers (E8).

A.4.2 Full-scale experiments

Below are instructions on conducting full-scale experiments to replicate the results in the paper. It also lists the expected outputs and how they support our claims. Note that if you have validated the functionality of the replication package

before the experiments, please reset the Docker container (remove, re-launch, and re-initialize it) to avoid data pollution caused by the results produced by the functionality validation process.

- **E1, E5, and E6.** [1 human-hour + 10 compute-day + 50GiB disk] Inside the Docker container, use the following command to synthesize the fuzzers using ELFUZZ or its variants (E6), or mine grammars and semantic constraints using the baselines:

  ```
  $ elfuzz synth -T <baseline> <benchmark>
  ```

  Please refer to the help message (`--help`) on what values `<baseline>` and `benchmark` can take. The following commands conduct the input generation experiments (E1 and E5):

  ```
  $ elfuzz produce -T <baseline> <benchmark>
  $ elfuzz minimize -T <baseline> <benchmark>
  $ elfuzz run rq1.seed_cov -T <baseline> <benchmark>
  $ elfuzz run rq3
  ```

  The coverage of the generated seed test cases will be recorded in an Excel sheet, which can be viewed via

  ```
  $ pyexcel view /elfuzz/analysis/rq1/results/seed_cov.xlsx
  ```

  The coverage of seed test cases generated by ELFUZZ is expected to be significantly higher than that generated by other methods (E1 results for C1). Similarly, you can view the coverage of seed test cases generated by ELFUZZ variants via

  ```
  $ pyexcel view /elfuzz/analysis/rq3/rq3_ablation.xlsx
  ```

  The coverage of seed test cases generated by ELFUZZ-NOSP, ELFUZZ-NOCP, and ELFUZZ-NOIN will show slight decreases, while that generated by ELFUZZ-NOFS will show significant decreases (E5 results for C3). The coverage trends of candidate fuzzers of the four variants during evolution can be viewed via

  ```
  $ pyexcel view /elfuzz/analysis/rq3/rq3_evolve_cov.xlsx
  ```

  The values will display curves similar to those shown in Figure 12 of the paper if being drawn (E6 results for C3).

- **E2 and E3.** [1 human-hour + 20 compute-day + 5GiB disk] The following command runs the mutation-based fuzzing experiments in RQ1 (E2) and collects and analyzes the results:

  ```
  $ elfuzz run rq1.afl --fuzzers <baseline_list> \
                       --repeat 10 <benchmark_list>
  ```

  The following commands will run the mutation-based fuzzing campaign in RQ2 (E3) and collect and analyze the results:

  ```
  $ elfuzz run rq2.afl --fuzzers <baseline_list> \
                       --repeat 10 <benchmark_list>
  $ elfuzz run rq2.triage
  ```

  You can view the averaged coverage trends during AFL++ fuzzing campaigns for RQ1 and trends of triggered bugs via

  ```
  $ pyexcel view /elfuzz/rq1/results/rq1_sum.xlsx
  $ pyexcel view /elfuzz/rq2/results/rq2_count_bugs.xlsx
  ```

  The data will show curves similar to those in Figures 8 and 9, i.e., the curves of ELFUZZ will be higher than that of others (E2 for C1 and E3 for C2).

- **E4.** [8 human-hour + 14 compute-day] The following command runs the real-world bug-finding experiment:

  ```
  $ elfuzz run rq2.real_world
  ```

  The triage and analysis of the results can only be done manually. The data tarball contains the bug-triggering test cases we found, though, in `rq2/results/cvc5_bugs`. You can check whether they actually trigger the bugs of the corresponding versions of `cvc5` (E4 for C2).

- **E7 and E8.** These two empirical case studies were done manually. The README file in the Docker image contains suggestions on how to replicate them (E7 and E8 for C4).

- **Reproducing the figures and tables.** Use the following command to reproduce the figures and plots presented in the paper:

  ```
  $ elfuzz plot --all
  ```

  Note that we cannot include proprietary fonts in the replication package. The appearance of the final rendered figures may be different from that in the original paper.

Please refer to the README files in the Zenodo repository, source code tarball, and Docker image tarball for more details.

A.4.3 Small-scale experiments

The full-scale experiments require a significant amount of time and substantial computational resources. Instead, you can choose to run some experiments at smaller scales, and the results can also support our claims. Below are instructions.

- **E1 and E5.** [1 human-hour + 1 compute-day + 25GiB disk] Unfortunately, decreasing the number of evolution iterations will significantly affect the performance of the synthesized fuzzers, which is expected. Thus, we cannot run the small-scale versions of the synthesis processes and E6. Therefore, we will utilize the original fuzzers included in the replication package. For E1 and E5, you can use the `--time 60` option for the `produce` subcommand to run the seed test case generation for 1 minute instead of 10. Even this short time limit should already demonstrate the advantage of ELFUZZ (C1) and the huge impact of excluding the fuzzer space from ELFUZZ (C3).

- **E2 and E3.** [1 human-hour + 5 compute-day + 5GiB disk] Note that using seed test cases generated in a time shorter than 10 min can weaken the advantage of ELFUZZ in AFL++ fuzzing campaigns, which is expected. Please reset the Docker container if you have run the above small-scale experiments before starting these two experiments, to ensure that you use the original test cases (generated in 10 minutes) included in the replication package. Then, you can change the value of the `--repeat` option from 10 to 3 to decrease the number of repetitions. Use the `--time 3600` option to run the AFL++ fuzzing campaigns for 1 hour instead of 24 hours. ELFUZZ should be able to show advantages within the first hour of fuzzing.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Rowhammer-Based Trojan Injection: One Bit Flip Is Sufficient for Backdooring DNNs

Xiang Li
George Mason University

Ying Meng
George Mason University

Junming Chen
George Mason University

Lannan Luo
George Mason University

Qiang Zeng*
George Mason University

A Artifact Appendix

This artifact demonstrates a novel inference-stage backdoor attack, ONEFLIP, which injects a backdoor into a full-precision model via a single bit flip. ONEFLIP achieves high attack success rates while causing minimal degradation to benign accuracy. Our work reveals a critical hardware-based vulnerability in DNNs: a highly effective backdoor can be injected into a full-precision model through a single bit flip.

A.1 Abstract

ONEFLIP consists of two stages: an offline stage and an online stage. The artifact includes all components required to reproduce the offline stage, which is sufficient to validate our results and verify our findings. For the online stage, we adopt the Rowhammer implementation from Blacksmith [1]. As our work does not introduce any new Rowhammer attack vectors, no additional disclosure is required.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact does not pose any risks to the evaluators' machine security, data privacy, or raise ethical concerns.

A.2.2 How to access

The artifact is publicly available on Zenodo at https://doi.org/10.5281/zenodo.15609595 to support replication. It includes detailed instructions for installation and execution, as well as examples of victim models and models backdoored by ONEFLIP.

A.2.3 Hardware dependencies

ONEFLIP is implemented on the NVIDIA H100 NVL GPU platform. We recommend using an NVIDIA GPU with CUDA support. However, a CPU-only environment can also suffice for reproducing key results.

A.2.4 Software dependencies

The software environment requires `Python` with the following packages: `PyTorch`, `torchvision`, `pandas`, and `numpy`.

A.2.5 Benchmarks

ONEFLIP is evaluated on four widely used datasets: CIFAR-10, CIFAR-100, GTSRB, and ImageNet. Then, we choose popular DNN architectures for image classification tasks for the datasets. For CIFAR-10, we adopt ResNet-18; For GTSRB, we adopt VGG-16; For CIFAR-100, we adopt PreAct-ResNet-18; and for ImageNet, we adopt ViT-base-16.

A.3 Set-up

We provide detailed steps to replicate our work in the README file. We recommend that evaluators refer to these materials for a better replication experience.

A.3.1 Installation

The artifact provides a `requirements.txt` file listing all necessary software packages and version information. The evaluators can create a new virtual `Python` environment and install the software dependencies:

```
pip install -r requirements.txt
```

A.3.2 Basic Test

Navigate to the artifact's main directory and execute the following command to run a basic test:

```
python test_attack_performance.py \
    -dataset CIFAR10 \
    -backbone resnet
```

*Corresponding author.

[1] https://github.com/comsec-group/blacksmith

A.4 Evaluation workflow

A.4.1 Major Claims

Here are the major claims made in our paper.

(C1): ONEFLIP enables one-bit-flip backdoor injection with high attack success rate (ASR) and minimal benign accuracy degradation (BAD). We use **E1** to verify **C1**. We provide a clean ResNet-18 trained on CIFAR-10, and one representative ONEFLIP-backdoored model for direct evaluation.

(C2): ONEFLIP can generate multiple backdoored model variants from a clean model, each achieving high ASR and minimal BAD. We use **E2** to verify **C2**. We take CIFAR-10/ResNet-18 as the representative combination to verify the experimental results in Section 6.3 and Table 3 of the main paper: ONEFLIP achieves state-of-the-art attack performance, reaching an ASR of 99.96% and a BAD of 0.01% with a single bit flip on CIFAR-10/ResNet-18.

A.4.2 Experiments

(E1): [ONEFLIP's capability of injecting backdoors via one-bit flip] [5 human-minutes + 5 compute-minutes + 5GB GPU memory]

Preparation: Ensure all dependencies listed in `requirements.txt` are installed. Then navigate to the main directory of the artifact.

Execution: Run the following command to evaluate ONEFLIP on the clean ResNet-18 trained on CIFAR-10 and one representative backdoored model we provide; the provided models are already placed in the corresponding directory:

```
python test_attack_performance.py \
    -dataset CIFAR10 \
    -backbone resnet
```

Results: The terminal will first display the benign accuracy (BA) of the original (clean) ResNet-18 model on the CIFAR-10 test set, which is expected to be approximately 86.43%. It will then show the selected flipped weight value at position (58,6) in the classification layer, before and after the bit flip, along with their corresponding 32-bit IEEE-754 binary representations (from `0 01111011 10110101000110001011000` to `0 01111111 10110101000110001011000`). Finally, it will print the BA of the backdoored model (expected to be approximately 86.40%) and the ASR (expected to be 100%), on the CIFAR-10 test set. These results confirm that ONEFLIP can inject a backdoor via a single bit flip with high ASR and minimal BAD.

(E2): [ONEFLIP's attack performance on CIFAR-10/ResNet-18 combination] [10 human-minutes + 2 compute-hours + 15GB GPU memory]:

Preparation: Ensure all dependencies listed in `requirements.txt` are installed. Then navigate to the main directory of the artifact.

Execution: Run the backdoor injection command to generate multiple ONEFLIP-backdoored variants from the provided clean ResNet-18 trained on CIFAR-10:

```
python inject_backdoor.py \
    -dataset CIFAR10 \
    -backbone resnet
```

All backdoored models generated by ONEFLIP will be saved to: `saved_model/resnet_CIFAR10/backdoored_models/clean_model_1/` for further evaluation. The filenames indicate the location of the one-bit flip, as well as the BA and ASR on the first batch of the CIFAR-10 test set (used as the attacker's clean sample set).

Next, run the following evaluation command to assess ONEFLIP's attack performance across all backdoored variants:

```
python test_attack_performance.py \
    -dataset CIFAR10 \
    -backbone resnet
```

Results: When running `inject_backdoor.py`, the program will first identify all eligible weights and filter them using the *degrad_threshold* to construct the potential weight set, as detailed in Section 5.3 and Algorithm 1 of the main paper.

Next, compute the neuron set based on the potential weight set, as all weights that share the same neuron in the feature layer can use the same trigger. For each neuron in this set, optimize a corresponding trigger pattern and report the mean output value of the neuron on the first batch of the CIFAR-10 test set embedded with the optimized trigger. Finally, evaluate the ASR on the same batch. Based on the *attack_threshold* parameter, all valid backdoored variants are filtered and saved. This procedure is detailed in Section 5.4 and Algorithm 2 of the main paper.

When running `test_attack_performance.py`, the terminal will first report the BA of the original (clean) ResNet-18 model on the CIFAR-10 test set, expected to be approximately 86.43%. Then, for each backdoored variant, it will output the bit-flip details, including the weight value and its corresponding 32-bit IEEE-754 binary representation before and after the bit flip, along with the BA and the ASR on the CIFAR-10 test set. All results are saved in a `.csv` file. A `Python` file `process_results.py` is provided to process this `.csv` file:

```
python process_results.py \
    -dataset CIFAR10 \
    -backbone resnet
```

The terminal will report the average BAD caused by ONEFLIP (expected to be approximately 0.01%) and the average ASR (expected to be approximately 99.96%). The results can verify the experimental results we provide in Section 6.3 and Table 3 of the main paper: ONEFLIP achieves state-of-the-art attack performance, reaching an ASR of 99.96% and a BAD of 0.01% with a single bit flip on CIFAR-10/ResNet-18.

A.5 Notes on Reusability

We believe ONEFLIP can be generalized to more dataset/model combinations and broader classification tasks. Specifically, the user needs to locate an eligible weight in the classification layer of the target model. Then, by obtaining the output of the last feature layer (e.g., via a hook function), the user can optimize a trigger to increase the output value of the neuron connected to the selected weight. Finally, by following the procedure described in Section 5.5 of the main paper to activate the backdoor, the attack can be successfully executed.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: From Purity to Peril: Backdooring Merged Models From "Harmless" Benign Components

Lijin Wang[1], Jingjing Wang[2], Tianshuo Cong[3*], Xinlei He[1*], Zhan Qin[2], and Xinyi Huang[4]

[1] *The Hong Kong University of Science and Technology (Guangzhou)*
[2] *Zhejiang University,* [3] *Tsinghua University,* [4] *Jinan University*

A Artifact Appendix

A.1 Abstract

In this paper, we provide the source code of MergeBackdoor and provide detailed instructions on how to implement MergeBackdoor using the provided code and evaluate its performance in different merging scenarios.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our work points out the potential threat of backdooring merged models. As our attack method is stealthy yet effective, it will be more dangerous if malicious users discover this attack. Our paper points out the problem and asks the whole community to pay more attention to it. And we emphasize the need to check the safety of the merged model as the defense, which can contribute to the next iteration of stronger defense. Since our experiments are conducted on publicly available datasets and models, they are non-destructive in nature.

A.2.2 How to access

The code is available now via zenodo link[1] with evaluation data[2] and the pre-trained models[3] by MergeBackdoor.

A.2.3 Hardware dependencies

The hardware conditions for our experiments are as follows: Our experiments use the Intel Xeon Platinum 8369B processor, which has 2 sockets, 32 cores per socket, a total of 128 threads, and we use 2 NVIDIA L20 GPUs each with 40GB of VRAM to run the experiments.

A.2.4 Software dependencies

We show detailed information in the following:

A.2.5 Benchmarks

All the datasets and models used in experiments are commonly available. They can be easily downloaded through the references cited in the paper. Most of the datasets and models can be obtained via the HuggingFace platform, while a small portion of the data (MLBD (mango) dataset) can be downloaded through the links provided in the original articles.

A.3 Set-up

We have provided detailed instructions on how to set up in the README.md of the source code.

A.3.1 Installation

Similar to the setup process, users only need to install the dependencies listed in the requirements.txt in the source code to run and verify the experiment. To facilitate verification, we have also provided the trained models for download.

A.3.2 Basic Test

We provide the code of eval_[merging_method].py and merge_adapaters_llms.py for the final validation experiments. After preparing the experimental models and data, you can conduct the validation experiments according to the instructions in the README.md.

A.4 Evaluation workflow

To validate MergeBackdoor, you need to prepare the dataset and the pre-trained model first. In our source code, we

* Corresponding authors: Xinlei He (xinleihe@hkust-gz.edu.cn) and Tianshuo Cong (congtianshuo@tsinghua.edu.cn)

[1] https://zenodo.org/records/14738608
[2] https://zenodo.org/records/14760016
[3] https://zenodo.org/records/14738289

have set CIFAR10 and MNIST to be automatically downloaded without manual intervention. We have also written the loading and processing methods for these datasets. For other datasets, manual download is required. The pre-trained model does not need to be manually downloaded, as the download process is already integrated into the code. After preparing the dataset and the pre-trained model, users can train the target model for MergeBackdoor using finetune_mergebackdoor(_NLP).py or mbd_llm.py (for LLMs) and verify the effectiveness of MergeBackdoor using eval_[merging_method].py or merge_adapters_llms.py (for LLMs).

A.4.1 Major Claims

(C1): *The clean sample test accuracy of the upstream models trained by* MergeBackdoor *is comparable to that of models trained normally.*

(C2): *When the upstream model is used alone, it does not exhibit backdoor behavior for data with triggers. That is, the Attack Success Rate (ASR) is at the level of random guessing.*

(C3): *The accuracy of the model fused by* MergeBackdoor *on clean samples is comparable to that of models trained normally under the same merging settings.*

(C4): *After merging the upstream models trained by* MergeBackdoor*, the model predicts the target label for samples with triggers, resulting in a high Attack Success Rate (ASR).*

A.4.2 Experiments

(E1): *[Training for Foundation Models] [5 compute-hour for each pair of dataset]:*
Preparation: *See README.md*
Execution: *Run the file "finetune_mergebackdoor.py" for ViTs and "finetune_mergebackdoor_NLP.py" for BERTs.*
Results: *The final models fine-tuned saved in the checkpoints as the name of "ViT(Bert)-[model1_name]-mdb.pth" and "ViT(Bert)-[model2_name]-mdb.pth". The ACC and ASR of these models and their average merging can be seen as the output of the console.*

(E2): *[Evaluation on Foundation Models] [50 compute-hour]:*
Preparation: *Get the fine-tuned models from E1.*
Execution: *To evaluate different merging methods, run the evaluation code file "eval_[merging_method].py" for ViTs and "eval_[merging_method]_NLP.py" for BERTs.*
Results: *The output will report all metrics of the merged model as in **Table 1** and **Figure 3** of the original paper.*

(E3): *[Training for LLMs] [24 compute-hour for each dataset]:*
Preparation: *See README.md.*
Execution: *Run the training file "mbd_llm.py" for LLMs.*
Results: *The final models are saved your specified saving folder with two directories. Each directory represents one fine-tuned adapter.*

(E4): *[Evaluation on LLMs] [50 compute-hour]:*
Preparation: *Get the fine-tuned models from E3.*
Execution: *To evaluate different merging methods, run the evaluation code file "merge_adapters_llms.py" or run "run_multi_merge.py" to evaluate all merging methods in one run.*
Results: *The output will report all metrics of the merged model as in **Table 2** and **Tabale 3** of the original paper.*

(E5): *[Evaluation of Multi-model Merging] [5 compute-hour for each pair of dataset]:*
Preparation: *Get the fine-tuned models from E1 and clean models uploaded.*
Execution: *After you have selected all the datasets, you can run files "eval_[model1_name]_multi_vit(bert).py" for ViTs and BERTs.*
Results: *The output will report all metrics of the merged models as in **Table 4** of the original paper.*

A.5 Notes on Reusability

We have publicly released the models trained by MergeBackdoor to facilitate the validation of experiments. Additionally, for situations with limited computational resources, we found that the Ties Merging experiment can be very time-consuming due to the large potential parameter search space. Therefore, we suggest that when computational resources are limited, users can increase the interval of parameter search (or use dynamic parameter search methods) to reduce the time overhead.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Revisiting Training-Inference Trigger Intensity in Backdoor Attacks

Chenhao Lin, Chenyang Zhao, Shiwei Wang, Longtian Wang, Chao Shen, Zhengyu Zhao*
Xi'an Jiaotong University

A Artifact Appendix

A.1 Abstract

In our artifact, we provide the source code along with a dataset archive. Specifically, the codebase consists of implemented algorithms for backdoor attacks and defenses,shell scripts that facilitate the TITIMworkflow, and analysis scripts to reproduce experimental results (*e.g.*, the heatmaps presented in Figure 6). The dataset archive includes four clean datasets mentioned in our paper, with clearly divided training, test, and validation splits, as well as their associated metadata (*e.g.*, backdoor configuration).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We admit that the proposed attack strategy under train-inference intensity mismatched triggers can be used for malicious purposes, however, it is unlikely to cause severe harm to individuals or society, since selecting the optimal trigger for each setting would require exponentially more resources than training. Moreover, the phenomenon of train-inference intensity mismatch can help form a foundation for developing more robust defenses against backdoor attacks, which can be beneficial to the machine learning community in the long term.

A.2.2 How to access

The source code and datasets are available on GitHub (https://github.com/cv12ha0/TITIM) and Zenodo (https://zenodo.org/records/14729436).

A.2.3 Hardware dependencies

Experiment with ResNet18 on CIFAR-10 require at least 8GB of RAM and a 2GB NVIDIA GPU. For other models and datasets, we recommend a machine with Linux operating system with 128GB of RAM and a 24GB NVIDIA GPU. In our experiment setup, we use a server running Ubuntu 22.04.4 with 2× Intel Xeon Gold 6226R CPUs, 256GB of RAM, and 4× NVIDIA GeForce RTX 3090 GPUs.

*Corresponding Author

A.2.4 Software dependencies

Our experiments are primarily conducted using Python 3.8, PyTorch 1.12.1, and CUDA 11.3. Additional experiments confirmed compatibility with Python 3.9, PyTorch 2.5.1, and CUDA 12.2.

The Instagram filters used in the backdoor attack Styled require MagickWand, which can be installed by running the following command:

```
$ sudo apt-get install libmagickwand-dev
```

Please refer to the instructions in the *README.md* file in our repository to set up the environment.

A.2.5 Benchmarks

To reproduce the main results in our paper, we require four datasets: MNIST, CIFAR-10, GTSRB, and CelebA. You can either run the shell scripts to prepare the datasets automatically or manually download the preprocessed clean datasets from Zenodo (https://zenodo.org/records/14729436).

A.3 Set-up

A.3.1 Installation

Please follow these steps to install the software dependencies:

CUDA Installation: Please make sure a CUDA toolkit with a version later than 11.3 is properly installed on your machine. You can check the official website of CUDA Toolkit for instructions: https://developer.nvidia.com/

Anaconda Installation: You can use venv or Anaconda to manage your Python environments; however, we recommend Anaconda for its ease of use. Please check the official website of Anaconda for more details: https://www.anaconda.com/download

Environment Creation: You can either create the running environment based on the *requirements.txt* or manually run commands in *scripts/env.sh* which contains exactly the same commands we used to set up our environment. Please refer to the *README.md* for details.

A.3.2 Environment Test

Run the following commands in shell to validate the environment:

```
$ python
>>> import torch
>>> torch.cuda.is_available()
```

The dependencies are installed properly if the output is "True".

A.3.3 Dataset Preparation

Activate the Conda environment and run commands in *scripts/get_clean_datasets.sh* to prepare the clean datasets automatically. You can also manually download the preprocessed clean datasets from Zenodo (https://zenodo.org/records/14729436) if you encounter network issues while running the scripts.

If downloaded properly, there should be a directory named "data/", which contains a clean subset for each specified dataset.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Models implanted with low training intensity triggers can generalize to higher-intensity inference triggers, while those with high training intensity triggers tend to overfit to higher-intensity inference triggers. This is proven by the experiments(E2) in Section 4.2 and Appendix D.

(C2): Attackers can improve the ASR by mixing triggers with varying intensities during data poisoning. For instance, one can improve the worst-case from approximately 10% (random guess) to at least 66.98% and exceeding 90% (92.77%) in extreme cases. This is proven by the experiments(E3) in Section 4.3 and Appendix C.

A.4.2 Experiments

(E1): [TITIM workflow] [5 human-minutes + 4 compute-hours + 6GB disk]: Run TITIM workflow to evaluate a single configuration (an attack with a specified intensity) and generate a heatmap as in Section 4.2.

Preparation: Download the clean datasets, create and activate the Conda environment as mentioned in Section A.3.

Execution: Run the full workflow in one script:

```
$ sh scripts/badnets_square.sh
```

Alternatively, you can run the scripts step by step.

Generate poisoned CIFAR-10 datasets by BadNets (Square):

```
$ sh scripts/inject/inject_badnets_square.sh
```

Train backdoored models:

```
$ sh scripts/train/train_badnets_square.sh 0
  cifar10 resnet18 0.05
```

Test backdoored models with varying intensities:

```
$ sh scripts/inference/crosstest_badnets\
  _square.sh 0 cifar10 resnet18
```

Draw a heatmap from the results:

```
$ python utils/scripts/tsv_reader.py
```

Scripts for other attacks are stored in the *scripts* folder, you can run them according to your specified attacks.

Results: The poisoned datasets are saved in "data/"; the backdoored models are stored in "res/"; the training and inference logs are saved as CSV files in "logs/"; the heatmap are generated and saved as PDF files in "heatmaps/". The trends in the heatmap should reflect the four regions described in Section 4.2 of the main text.

(E2): [Intensity mixture] [5 human-minutes + 7 compute-hour + 5GB disk]: Train backdoored models with two intensities by BadNets(Square) and generate a heatmap as in Figure 21.

Preparation: Download the clean datasets, create and activate the Conda environment.

Execution: Run the full workflow in one script:

```
$ sh scripts/badnets_square_mix.sh
```

Alternatively, you can run the scripts step by step.

Generate poisoned datasets by BadNets (Square) with two opacities:

```
$ sh scripts/mixtest/inject_mix.sh
```

Train backdoored models:

```
$ sh scripts/mixtest/train_mix.sh
```

Inference with varying intensities:

```
$ sh scripts/mixtest/crosstest_mix.sh
```

Draw the heatmap:

```
$ sh scripts/mixtest/draw_mix.sh
```

Results: The poisoned datasets are saved in "data/"; the backdoored models are stored in "res/"; the training and inference logs are saved as CSV files in "logs/"; the heatmap are generated and saved as PDF files in "heatmaps/".

A.5 Notes on Reusability

The modular approach of this artifact ensures compatibility and facilitates collaboration across different projects and implementations. For instance, as long as the dataset is packaged as a DataFrame with PNG files and labels, along with the model and backdoor algorithms adhering to the defined interface specifications, our work can be easily reused. Please refer to the repository for more details.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: HyTrack: Resurrectable and Persistent Tracking Across Android Apps and the Web

Malte Wessels, Simon Koch, Jan Drescher, Louis Bettels, David Klein, Martin Johns
Technische Universität Braunschweig
{malte.wessels, simon.koch, jan.drescher, louis.bettels, david.klein, m.johns}@tu-braunschweig.de

A Artifact Appendix

A.1 Abstract

Android apps can freely intermix native and web content using Custom Tabs and Trusted Web Activities. This blurring of the boundary between native and web, however, opens the door to HyTrack, a novel tracking technique. Custom Tabs and Trusted Web Activities have access to the default browser state to enable, e.g., seamless reuse of authentication tokens. HyTrack abuses this shared browser state to track users both in-app and across the web using the same identifier. While we do not find direct evidence that our technique is already employed, our findings indicate that all essential components are currently in place.

This artifact contains a demo with description, the proof-of-concept apps and backend, as well as our tools for our three measurements: Static app analysis, dynamic network analysis of apps, and a web measurement.

A.2 Description & Requirements

Since the dynamic app network analysis requires rooted devices and an intricate network setup, we exclude it from the functionality artifact evaluation. However, our artifact contains the specific version of the pipeline for posterity. It is based on a previous publication at USENIX with minimal changes. Our version is functional but not within the scope of this artifact evaluation. Refer to that previous publications artifact appendix[1], the READMEs in our artifact, as well as the current development version[2] for more information.

A.2.1 Security, privacy, and ethical concerns

Since HyTrack is a Proof-Of-Concept implementation of a new tracking technique, it tracks users by design. Make sure that the general public does not use PoC instances. Since it is a PoC implementation, we do not guarantee security. However, it is neither destructive nor a security risk by design.

The web scraper will request the specified hosts, and the static analysis pipeline will decompile the provided APKs.

A.2.2 How to access

The artifact is available on Zenodo at https://doi.org/10.5281/zenodo.14718794.

A.2.3 Hardware dependencies

Our PoC can be run on real Android phones. However, we rely on the Android emulator to streamline the testing for this artifact evaluation. Therefore, we require a capable x86 host system with hardware-accelerated emulation and a few gigabytes of RAM and storage for the Android guest. Hardware acceleration can be turned on in the host's UEFI, and it is called Intel Virtualization Technology (VT-x, vmx) or AMD-V (SVM)[3].

Additionally, we require about 20 GB of Docker images that will be pulled or built.

A.2.4 Software dependencies

A modern Linux system with docker, docker-compose, and Python 3. An Android toolchain with the commands adb, emulator, avdmanager, and sdkmanager must be available. Refer to your distros resources or Android Studio to install them.

A.2.5 Benchmarks

The app analysis requires an (any) Android app as a single APK file to be run on. An easily accessible source for free APKs is fdroid [4]. The web scraper requires a list of hosts to scrape. However, it is bundled with one exemplary list.

[1] https://www.usenix.org/conference/usenixsecurity23/presentation/koch
[2] https://github.com/App-Analysis/scala-appanalyzer
[3] https://developer.android.com/studio/run/emulator-acceleration
[4] https://f-droid.org/en/packages/

A.3 Set-up

A.3.1 Installation

Emulator Setup Create a new Android virtual device (AVD). Ensure the image is available by running `sdkmanager "system-images;android-30;google_apis_playstore;x86"`. Create the device via `avdmanager create avd -n hytrack0 -k "system-images;android-30;google_apis_playstore;x86" --device "Nexus 5"`. Start the device via `emulator @hytrack0`. Note that this blocks this terminal window.

Mobile Chrome Setup Wait for the device to boot and launch Chrome. Click through its first-use setup. Trusted Web Activities used by HyTrack require a Digital Asset Link. Regularly, Chrome only accepts HTTPS to fetch Digital Asset Links We must disable this for local testing[5]: In the mobile Chrome, navigate to `chrome://flags`. Search for and enable "Enable command line on non-rooted devices". Restart Chrome as prompted. On the host, run `adb shell "echo '_ --disable-digital-asset-link-verification-for-url=\"http://10.0.2.2\"' > /data/local/tmp/chrome-command-line"`.

Android Build Container We can use a docker container to build the apps and avoid conflicts with your host system. This requires a small setup step, as the apps are signed with a release key. In the PoC subfolder, run `docker run -it -v `pwd`:/in runmymind/docker-android-sdk:ubuntu-standalone-20250210 --entrypoint=bash`. This will launch a shell in the container. To generate signing keys for testing run `keytool -genkey -v -keystore /root/testdeploykey -keyalg RSA -alias key0`. Use 'password' as the password. Skip all questions with Enter, and confirm the empty selections with a 'yes'. Don't close this container's terminal until you have built the apps by completing the following steps. The working directory is mounted inside the container at `/in`.

A.3.2 Basic Test

1. To test the PoC web server, it is sufficient to run `docker-compose up` on the host system in the PoC/webapp directory and to navigate to localhost in your favourite browser. A page is loaded.

2. To test the PoC apps (CrossAppLauncher, CrossAppTrackerOne), compile them with the build container: In the container's terminal, cd into one app's folder at `/in` and run `./gradlew assemble`. Repeat with the other app.

3. To test the webscraper, run `docker build . -t wk-crawler:latest` in the webscrape folder.

4. To test the static app analysis, run `docker-compose --env-file .env up -d` in `staticanalysis/analysis/pipeline/`.

5. Ensure that the demo folder contains a playable video and a description file.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *HyTrack can track users across the web and native applications. It can restore itself from browser storage resets, and is seamless.*

(C2): *We provide toolchains to search for indications of HyTrack in the wild.*

A.4.2 Experiments

(E1): *[Demo reenactment] [20 human-minutes]: Reenacting the demo shows HyTrack capabilities.*

How to: Watch the demo video and reenact it in the emulator. Note that in HyTrack threat model, a tracking provider hosts its backend on a domain. In the paper, we've used the generic domain ad.com. For the demo, we hosted the backend on the URL `examplecorp.de`. Replace that with `http://10.0.2.2` in the emulator, which maps to localhost port 80 on your host system. The other tracking web page maps to `http://10.0.2.2:8080` in the emulator.

The 20 minutes includes getting familiar with the video and setup. The demo itself can be conducted in two minutes.

Preparation: With a running emulator, web server, and compiled apps, as described in the Set-Up and Basic test sections, install both apps to the emulator with `adb install app-release.apk` in the `app/build/outputs/apk/release/` directories.

Execution: Watch the demo video in the Demo folder, read the demo description in the file next to it, and reenact it step-by-step in your emulator setup. Adjust the domains as described above. Instead of clicking on shortcuts to URLs, please type them in Chrome's URL bar. The 'clear storage' option might be moved to 3-Dots-Menu -> Settings -> Privacy.

Results: A similar behavior to the video should be achieved. Some minor differences can occur: Due to the debugging mode, Chrome displays a warning about enabled flags, which can be dismissed. Depending on the Chrome version present on the virtual device, the 'Running in Chrome' notice and position of the 'clear storage' options might be different.

[5] https://developer.chrome.com/docs/android/trusted-web-activity/integration-guide#debugging

(E2): *[Webscrape] [10 human minutes]*

How to: Relates to C2: To demonstrate our web scraper, which scrapes the DAL from websites, we run it on three websites. DALs are required to run TWAs, one component used by HyTrack.

Preparation: `docker build . -t wk-crawler:latest` in the webscrape folder.

Execution: To scrape google.com, wikipedia.org, and youtube.com for DALs, run `docker run -v $PWD/resources:/resources wk-crawler:latest -endpointsFilePath=/resources/dal.txt -websitesFilePath=/resources/test_top-5000.csv` in the webscrape folder. Afterward, copy the results out of the docker container: `docker cp containerName:/proj/wellKnownCrawlerMain/crawl_results results`. Replace `containerName` with the container name. It can be found via `docker ps -a | grep wk-crawler`.

Results: A results folder, containing subfolders. The `endpointInformation` subfolder contains a (potentially hidden) `.well-known` subfolder, containing a subfolder for assetlinks, which includes a JSON. This should contain results for the asset links found.

(E3): *[Static App Analysis] [10 human minutes]*

How to: Relates to C2: We will run the static analysis pipeline on one application to demonstrate its functionality. It searches for features that could be used to implement HyTrack or its components.

Preparation: Download one APK[6] in a new folder `staticanalysis/analysis/pipeline/apks`. Run `python analysis/pipeline/folder2json.py apks/` in that folder to generate the file `apps.json`. This file is the input for the static pipeline.

Execution: Run `docker-compose --env-file .env up -d` to prime the pipeline. If you get a port blocked error, quit the emulator or any running ADB process, as ADB defaults to the same port. Run `docker-compose --env-file .env --profile init up -d` to start the analysis.

Results: A JSON file in the `analysis/pipeline/results` folder. It contains many keys for CT features, TWA features, extracted DALs, and schemes. For the example app, all entries are empty/false except for schemes, which contain 'HTTP' and 'HTTPS'.

A.5 Notes on Reusability

This work and our HyTrack-PoCs are intended as an early warning sign, we assume and hope that the issue is fixed systematically in the future. The PoCs could then be used to validate the fixes.

The dynamic app analysis pipeline is actively maintained. If you intend to use it for your work, check out its latest version on GitHub[7] to profit from all future fixes and features. The static app analysis is a fork from previous work by Beer at al. which is available at https://github.com/beerphilipp/tabbed-out.

Refer to the READMEs in the subfolders for more details.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

[6] https://f-droid.org/repo/cz.martykan.webtube_17.apk

[7] https://github.com/App-Analysis/scala-appanalyzer



USENIX Security '25 Artifact Appendix: Seeing Through: Analyzing and Attacking Virtual Backgrounds in Video Calls

Felix Weissberg*
BIFOLD & TU Berlin

Jan Malte Hilgefort*
TU Braunschweig

Steve Grogorick
TU Braunschweig

Daniel Arp
TU Wien

Thorsten Eisenhofer
BIFOLD & TU Berlin

Martin Eisemann
TU Braunschweig

Konrad Rieck
BIFOLD & TU Berlin

A Artifact Appendix

A.1 Abstract

Video calls have become an essential part of remote work, but transmitting video from home risks exposing private details. To address this, video conferencing platforms provide *virtual backgrounds* to conceal the real environment. Unfortunately, this protection is not flawless, and occasional pixel leak from the environment. In this paper, we introduce a reconstruction attack that restores the real surrounding of videos protected by virtual backgrounds. To evaluate the effectiveness of this attack, we develop a testing framework that generates a set of videos with different virtual backgrounds and caller environments based on recordings made in front of a green screen. Within this framework, we use two virtual background implementations from common video conferencing services–MediaPipe (Google Meet, Jitsi, BigBlueButton) and Zoom. We further implement two baselines attacks from Sabra et al. and Hilgefort et al. that serve as a reference for the reconstruction quality of our attack and find that our reconstructions reveal at least 53% more leaked pixels from a video.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no expected risks or others ethical concerns when executing the artifact.

A.2.2 How to access

We provide our artifact as a stable archive on Zenodo as well as on Github at commit state `0d1238a0c43e4910d62683ff06a2553da35568ab`.

A.2.3 Hardware dependencies

The artifact does not require specialized hardware, but a GPU can significantly reduce the total execution time.

*Authors contributed equally.

A.2.4 Software dependencies

The artifact requires *apptainer* and *curl* to be installed on the system. Detailed instructions on how to install apptainer on common Linux distributions can be found in the apptainer documentation. The curl package can be installed via the package repository on common Linux distributions.

A.2.5 Benchmarks

Models. The artifact requires the U^{2}Net model to execute the baseline attack from Hilgefort et al. as well as the DeepLabV3 model for the approach from Sabra et al. and our attack. Both models are downloaded automatically by the evaluation scripts.

Data. Furthermore, green screen recordings are required from which the evaluation dataset is constructed. We adhered to best practices when collecting the video recordings for our evaluation. As part of the privacy policy agreed upon by the participants, this included limiting the use of the recordings strictly to the minimum required to conduct the evaluation. This policy also ensures that all recordings are deleted at the latest three years after the recording. Consequently, we do not release the participants' videos. Instead, we provide a sample recording that allows to reproduce the attack's effectiveness. This recording is already contained in the archive provided via Zenodo as well as in the Github repository.

Code. We provide the code necessary to re-run the attack as well as the two approaches from Hilgefort et al. and Sabra et al., and the code for the evaluation of their respective reconstruction performances. However, as described in our *Open Science* statement, due to restrictions in Zoom's terms and conditions, we cannot share the tool used to extract portrait masks from the Zoom client. As a remedy, we directly provide the extracted masks to facilitate reproducing the attack without additional reverse-engineering efforts. These masks are also included in the Zenodo archive and the Github repository.

A.3 Set-up

A.3.1 Installation

The artifact can be obtained as a zip compressed archive on Zenodo or cloned from the Github repository. Provided the necessary software dependencies described in Appendix A.2.4 are installed and the artifact was acquired in either of the two ways, the `scripts/build.sh` script can be run from the artifacts' root directory to set up the environment.

A.3.2 Basic Test

To verify that the setup works without running the attack or any baseline approaches, the script `scripts/check.sh` can be executed from the artifacts' root directory. A successful test ends printing: `[*] Success!`.

A.4 Evaluation workflow

A.4.1 Major Claims

In this paper we introduce a novel approach to reconstructing real surroundings in video calls where the environment is concealed with a virtual background. We compare our approach to two previous works from Hilgefort et al. and Sabra et al. and claim that our proposed attack significantly outperforms them with a reconstruction performance that is at least 2.64 and 1.83 times higher, respectively.

We substantiate this claim in Section 5.3 in which we conduct a quantitative assessment of the reconstruction performance of the individual approaches and show our results in Table 3 in the paper.

A.4.2 Experiments

In order to assess the performance of the reconstruction approaches, we generate videos of individuals in video calls featuring diverse surroundings and virtual backgrounds. These videos are created using recordings in front of a green screen, providing ground-truth masks of the caller. These masks enable the assessment of pixel leakage from the real surroundings in each frame. In combination with the masks created by the video conferencing services to insert the virtual background, this allows for a perfect reconstruction of the surroundings, serving as a reference for evaluating the attacks.

We expect the evaluation to require approx. 10 human minutes and 48 hours on consumer hardware (recent CPU with 16 cores and 32GB RAM) and no GPU acceleration.

Preparation. Make sure the necessary dependencies are installed as described in Appendix A.2.4 and the artifact is acquired and setup as described in Appendix A.3.1. That is, it should be downloaded and unpacked from Zenodo or cloned from the Github repository and the script `scripts/build.sh` should have been executed successfully.

Execution. To run the experiment starting from the dataset generation over conducting our attack as well as the two baseline approaches, to finally evaluating the reconstruction performance, the script `scripts/run.sh` has to be executed.

Results. Upon successful termination of the script, the reconstruction performances are calculated as described in paragraph *Measuring reconstructions* in Section 5.3 of the paper and printed on the screen for each individual attack with `vader` referring to the performance of our approach, `hilgefort` to the one from Hilgefort et al. and `sabra` to performance of the attack by Sabra et al. The results are shown per video conferencing system with `zoom` indicating the performance for videos that use the virtual background feature from Zoom and `mp` for MediaPipe. The scores should be close to the following values:

```
Attack: vader
vader-mp-interview: mean 0.1546
vader-zoom-interview: mean 0.1643
Attack: sabra
sabra-mp-interview: mean 0.0595
sabra-zoom-interview: mean 0.0691
Attack: hilgefort
hilgefort-mp-interview: mean 0.0093
hilgefort-zoom-interview: mean 0.0038
```

The reconstruction performance is significantly higher for our attack compared to the two other considered approaches substantiating our claims in the paper (see Appendix A.4.1).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: "Endangered Privacy: Large-Scale Monitoring of Video Streaming Services"

Martin Björklund, Romaric Duvignau
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
martibjo@chalmers.se, duvignau@chalmers.se

A Artifact Appendix

This artifact appendix guides the reader into reproducing the results (i.e., plots) of our paper. Since we provide the raw network capture and the entire tool chain to reproduce our evaluation, all claims contained in our work can be reproduced with our artifact.

A.1 Abstract

Our work presents a fingerprinting-based attack on streaming platforms where the attacker is shown to be capable of rapidly and accurately identifying any video from the streaming service, based on capturing network traffic (either as a "strong attacker" by having access to the network or by eavesdropping a wifi connection aka the "weak attacker" model).

Our entire fingerprint dataset (covering the streaming platforms *Amazon Prime Video*, *Max* and *SVT Play*) and all the needed tools to run the attack are made publicly available in our Zenodo permanent artifact[1], which includes:

- TSV datasets (in `datasets/tsv/`) containing video metadata and fingerprints from Amazon, Max, and SVT;
- karl (Go source files `software/karl/`, binary `karl-x86_64-linux`), our data collection tool for retrieving fingerprints;
- BurstShark (Rust source files `software/burstshark/`, binary `burstshark-x86_64-linux`), our TShark wrapper tool for network traffic capture;
- Models (Python modules `software/models.py` and `software/datasets.py`) implementing the video identification logic.

In addition, our artifact also contains the data that we have used during our evaluation:

- PCAP capture files from the main evaluation (in `datasets/pcaps/`), the unknown traffic experiment (in `datasets/unknown-pcaps/`), and DAITA testing (in `datasets/daita-pcaps/`);
- An evaluation script (Python script `software/evaluate.py`) to reproduce the main results;
- Miscellaneous scripts (in `software/misc/`), e.g., the automated WebDriver streaming script (`software/misc/webdriver-stream/main.py`).

At last, note that tables appearing in our evaluation are simply summarization of results that our evaluation script also produces. We also provide the pcaps we used with Mullvad's DAITA service activated (in `datasets/daita-pcaps`) that we mention in § 5 "Mitigation" as a preliminary evaluation.

In addition, to speed-up the reproducibility of all our results, we provide preprocessed pcap data in `software/precomputed/`. This data allows calculating the results of the identification of the 900 videos played during our evaluation for one set of parameters within a few minutes. The preprocessed data can also be reproduced as well but requires longer time and larger memory requirement as the 3 k-d trees are stored in memory during the evaluation (following the weak attacker model).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We hereby certify that all our tools are harmless. The source code of the three developed programs is shared and is short to verify (under a few hundred lines of codes for each tool):

- for our network capture program BurstShark, in the `software/burstshark` folder (written in Rust) for the precompiled binary `burstshark-x86_64-linux`;
- for our fingerprinting data collection tool karl, in the `software/karl` folder (written in Go) for the precompiled binary `karl-x86_64-linux`;

[1] https://doi.org/10.5281/zenodo.14676526.

- in the `software/` folder for our Python identification tool: modules `datasets.py` and `models.py` and our evaluation script `evaluate.py`.

Disclaimer: One of our artifact's tool (karl) collects data directly from live services, which might contravene terms of service (ToS) of the targeted platforms. It's designed for research purposes and should be used with caution. Additionally, since the program interacts with live services, the functionality may break at any moment if the targeted services update their implementation.

A.2.2 How to access

Our artifact is permanently hosted on the Zenodo platform: https://doi.org/10.5281/zenodo.14676526.

A.2.3 Hardware dependencies

No special hardware is required to run our artifact, however the memory requirements are as follows:

- For running the evaluation based on **preprocessed pcaps**: a minimum of about **32 GB** is recommended. In our tests, memory usage of the evaluation script typically peaked at about 20 GB.
- For running the evaluation from the **raw pcap files** as well as running the identification tool (with a database of 240k videos for the 3 services, as in our evaluation) in live tests: a minimum of **192 GB** of memory is recommended.

A.2.4 Software dependencies

Our tools are dependent on the following libraries/programs:

- **Python packages** (main identification tool): joblib, numpy, pandas, pyarrow, python-dateutil, pytz, scikit-learn, scipy, six, threadpoolctl, tzdata; (web driver used in our automatized evaluation) attrs, certifi, charset-normalize, h11, idna, outcome, packaging, PySocks, python-dotenv, requests, selenium, sniffio, sortedcontainers, trio, trio-websocket, typing_extensions, urllib3, webdriver-manager, websocket-client, wsproto;
- **Python 3.12.4**;
- **tshark** for BurstShark.

Required python packages are listed in the corresponding requirement files.

In our artifact, we provide precompiled binaries for Linux X86_64 architecture. If one needs other architecture or wishes to compile the binaries from the source files, the standard **Go** and **Rust** compilation toolkit are necessary.

In this guide, we give command-lines for a Unix user, however, all our tools are also available on other OS. For running the time measurement tests (E1 in § A.4.2), the `bc` utility is used, it can be installed e.g. on Debian-based distributions with:

```
sudo apt-get install bc
```

Streaming Platform Subscriptions In this guide, all examples do not require any login information on the streaming platforms used. In particular, this concerns retrieving the fingerprint directly from a video's URL using our karl tool. To run our data collection tool for the Amazon Prime and Max platforms, one needs to set the required cookies by being authenticated on the streaming platforms, i.e., the authentication cookies ("ubid-main", "x-main", and "at-main" for Amazon; "st" for Max) are required so to access certain data.

A.2.5 Benchmarks

The only required datasets and models are the ones provided in our artifact and the VNAT public dataset.

A.3 Set-up

A.3.1 Installation

Download The README provided as part of our Zenodo artifact record describes how to setup our artifact:

1. Download[2] the two necessary archives from Zenodo's permanent record datasets.tar.gz (**23.8 GiB**, uncompressed 97.2 GiB) and software.tar.gz (**5.9 GiB**, uncompressed 7.9 GiB) and decompress both archives in the same directory, i.e., run the following commands:

```
mkdir endangered-privacy
cd endangered-privacy
wget https://zenodo.org/records/15051891/files/
     datasets.tar.gz
tar -xvzf datasets.tar.gz
wget https://zenodo.org/records/15051891/files/
     software.tar.gz
tar -xvzf software.tar.gz
```

2. To rerun the full unknown traffic evaluation, the VNAT dataset is required; the dataset can be downloaded from MIT's webserver[3]. Download (**36 GB**) and extract the dataset from VNAT Dataset directly into the `datasets/unknown-pcaps/` directory (e.g., `datasets/unknown-pcaps/VNAT_release_1/`).

Installation (via conda) Here follows guided instructions for installing the required Python packages via conda[4] with

[2] For convenience, we provide here direct links to elements of version v5 of our artifact. If a newer version becomes available, we recommend to download the latest version of the artifact.

[3] https://www.ll.mit.edu/r-d/datasets/vpnnonvpn-network-application-traffic-dataset-vnat.

[4] https://www.anaconda.com/download/success.

a dedicated virtual environment for running the code. First conda must be installed, thus follow the corresponding guide for the system at hand[5]. For Linux-64, this is done as follows:

```
mkdir -p ~/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-
     latest-Linux-x86_64.sh -O ~/miniconda3/miniconda.sh
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
rm ~/miniconda3/miniconda.sh
```

Then, run the following commands:

```
conda create -n py3124 python=3.12.4
conda activate py3124
python -m ensurepip --upgrade
pip install --upgrade pip
pip install -r requirements.txt
pip install matplotlib
```

Now, you can check that you can run the evaluation script:

```
python evaluate.py --help
```

Installing TShark Install TShark following https://tshark.dev/setup/install/, e.g. on Ubuntu run:

```
sudo apt-get install tshark
```

Compile binaries from sources (optional) We use here the standalone binary `burstshark-x86_64-linux` for our tool BurstShark. If one needs to recompile it, use the source files from folder `burstshark` and the appropriate Rust tools.

The karl binary, `karl-x86_64-linux` included in the package, is our custom data collection tool. The source code is available in the `karl/` directory and can be compiled with Go tools.

A.3.2 Basic Test

Guided test of the data collection tool Let us first illustrate how we gather information and metadata about the videos we would like to fingerprint. Our data collection tool has two options `extract-urls` and `extract`, cf. running the tool with `-help` flag. Here is an example of such usage:

```
./karl-x86_64-linux extract https://www.svtplay.se/video
     /e9qaywe/leif-och-billy/1-en-ny-tjej-i-byn
```

Once karl has acquired the necessary metadata for a given video, the tool builds a fingerprint for each representation of the queried video using the information from its manifest file.

For instance, when the above command-line is executed, karl retrieves 10 fingerprints of the 10 representations (aka "qualities") associated with the queried video, with the downloaded data organized in a json (called `extract_<timestamp>.json`.) with the following format:

[5] https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html.


```
{
  "service": "svt",
  "url": "https://www.svt[...]billy/1-en-ny-tjej-i-byn",
  "videos": [
    {
      "id": "eQJRo6j",
      "title": "Leif \u0026 Billy - 1. En ny tjej i byn",
      "playback_url":
          ↪ "https://www.svtplay.se/video/eQJRo6j",
      "duration": 835,
      "expires_at": "2025-04-01T23:59:00+02:00",
      "variants": [
        {
          "mime_type": "video/mp4",
          "codecs": "hvc1.2.4.L123.90",
          "width": 1280,
          "height": 720,
          "bandwidth": 2875167,
          "fingerprint": {
            "segment_sizes": [
              812697,
              1337592,
              1274793,
              [...]
            ],
            "segment_durations": [
              49152,
              49152,
              49152,
              [...]
            ],
            "timescale": 12800
          }
        },
        [...]
```


karl can also build a fingerprint directly from a manifest file's URL, useful for fingerprinting arbitrary manifests/mp4 files; an example of such functionality can be tested by running the following command-line:

```
./karl-x86_64-linux fingerprint https://media.axprod.net
     /TestVectors/Cmaf/clear_1080p_h264/manifest.mpd
```

This generates a file `fingerprint_<timestamp>.json`, with the following json schema (`variant` schema is identical to `variants`'s format from extract's json):


```
{
  "url": "https://media[...]_1080p_h264/manifest.mpd",
  "variant": [
    {
      "mime_type": "video/mp4",
     [...]
[...]
```


As of March 2025, you can test the fingerprinting functionality on the 6 provided example manifest files listed in our `README.md`, covering DASH manifests, HLS manifests and Fragmented MP4.

A single entire video representation is typically gathered in the matter of 10-100ms, hence already demonstrating that our data collection **does not** require to watch the fingerprinted videos. Note that our karl tool is designed to make the process of mass collecting of such fingerprints very efficient: connections will be kept open, reused and parallelized. Also, if

no country code is explicitly specified, a small geolocation lookup subroutine is run once before downloading the fingerprint(s). SVT fingerprints generally take slightly longer because of explicit addressing requiring many requests, but even for those videos, our tool still runs in subsecond time per representation.

The "segment sizes" information is the one our identification tool uses. After running our tool over the 3 studied streaming platforms, these information have then been gathered into 3 datasets. You can check how a representation is encoded into our tsv database:

```
head -n 2 datasets/tsv/amazon_representations.tsv
```

Guided test of the identification tool For quick evaluations or experiments (the default setting), precomputed results can be used, significantly reducing the processing time (minutes) and system requirements.

The standard parameters for the evaluation script (that we have used in most of our evaluation) are:

- `-model strong,weak,unknown` – all models;
- `-method network,wifi` – all methods;
- `-theta THETA` – θ =2.2;
- `-time [1-600]` – 600;
- `-recompute` – this extra flag is not added at this stage so not to recompute tsv files from raw pcaps (cf. complete evaluation in § A.4.2);
- `-burstshark-path` – only if the binary is not `./burstshark-x86_64-linux`.

For example, to run a single model/network evaluation with a given θ, use:

```
python evaluate.py --model strong --method network --
    theta 1.5 --time 600
```

The expected output is as follows:

```
Loading dataset... DONE
Eval. model 'strong' with method 'network'... DONE
Saved ./results/strong-network-1.5-600.csv
Saved ./results/strong-network-1.5-600-aggregated.csv
```

Hence, this first test produces 2 "result" csv files saved in the `results` folder, with the non-aggregated file showing our identification results (successful identification and identification time) per played video (from a pre-preprocessed pcap capture):

```
device service video id[...] unknown mis[...] id_time
linux amazon amzn1.[...]2c56 1 0 0 49.15[...]
linux svt jwAypqm 1 0 0 59.9738812446594
linux max 899[...]448 1 0 0 60.25[...]
windows svt jaRm5gA 1 0 0 87.2628059387207
windows amazon amzn1.[...]70a2 1 0 0 72.60[...]
windows max e7[...]ccf6 1 0 0 70.41[...]
mac amazon amzn1.[...]f92d 1 0 0 61.4[...]
```

As an indication, on our recent but not high-end laptop (CPU: Intel Ultra 7 155U, 12 cores; 32GB RAM), it took about 3min to run that test with the program's memory usual usage at around 10GB and peaking at about 18-20 GB. Since this test is running over a fixed network capture and our identification system is deterministic, re-running the evaluation will produce identical results (note that `identified_time` refers to the packet capture time not the time of the identification system, which is negligible in comparison).

The aggregated results merge the results per platform, eg for $\theta = 1.5$, accuracy, precision and recall are about 98.3% for the Max service.

To run our complete evaluation script with the standard parameters, simply run:

```
python evaluate.py
```

This test took for us about 6min and test the strong attacker with "network" method as well as the weak attacker with both "network" and "wifi" methods. The unknown traffic test creates a result `unknown-scores.csv` file with the following simple format:

```
dataset application score
self hbomax 0.167802384471442
self hbomax 0.584277198585956
self hbomax 0.52584947872736
```

Since this shows the maximum score obtained by the most likely representation-timestamp from the fingerprint database, it suffices to check if any is above a given threshold θ to count "false positives"; the highest value for the maximum score is 1.998 in our experiments, hence any $\theta > 2$ will produce 0 false positive with this test data. Recall, our recommended and default value for θ is 2.2.

A.4 Evaluation workflow

A.4.1 Major Claims

The main claims of our work can be summarized as:

(C1): *It is fast for the attacker to build the necessary fingerprint database*. We provide our capture tool and show here how to run it with some examples. The required information for an entire video is typically acquired in less than 100ms per fingerprint, demonstrating the rapidity at which one can build the database. We publish our entire fingerprint database acquired by running the tool over the libraries of the three services, something easily reproducible with a laptop and an ordinary internet connection (some of the services require being authenticated).

(C2): *The proposed attack quickly shows high accuracy, regardless of the type of attacker*. This claim is supported by Figure 6 which can be reproduced by running our evaluation script over the PCAP captures that we have recorded during our experiments. In addition, the scripts

that were used to generate our network captures are also provided, hence one could also generate new PCAPs under the same conditions. Figure 8 can also be reproduced, simply by running the provided evaluation with different value for our threshold parameter θ.

(C3): *The false positive rate is very low*. This claim is supported by the distribution of maximum scores produced (Figure 7) by our system on the "unknown traffic". We provide the pcaps corresponding to HBO Max and our youtube captures in folder `datasets/unknown-pcaps/self` whereas the VNAT dataset is publicly available and should be added in `datasets/unknown-pcaps/` to run the script without modification. Our provided evaluation script directly calculates the scores of Figure 7 from the network captures.

A.4.2 Experiments

To check our claim C1:

(E1): *[Data-Collection] [15 human-minutes]: Test the provided tool for building the fingerprint dataset, and check time measurements of acquiring video fingerprints.*
Preparation: Follow the previous instructions on running the example tests to setup the system.
Execution: Run the commands with example URLs and manifest files following our artifact's README. Check gathered information and jsons and our fingerprint tsv databases. To test the speed of our data collection, download and execute the following speed tests from our artifact (this assumes the artifact was uncompressed in the current directory as per § A.3.1's instructions) – time per fingerprint is the displayed time divided by 10 (number of representations for the tested video):

```
wget https://zenodo.org/records/15051891/files/time-
    experiment.tar.gz
tar -xvzf time-experiment.tar.gz
cd time-experiment/
./time.sh 20 ../software/karl-x86_64-linux --out-
    dir time-test-svt extract https://www.svtplay.
    se/video/edPkzzZ/leif-och-billy/1-stromlost
```

Optionally, test also with more SVT videos and Amazon's time collection following the time experiments' README's instructions.
Results: Check the (mean, median, standard deviation) times it took for collecting the json fingerprints.

Claims C2-C3 can be assessed using the preprocessed pcaps following the test guide. One can also run our complete evaluation directly from raw pcaps (hence, running our burst generator over the recorded network traffic) as follows:

(E2): *[Complete-Evaluation] [30 human-minutes + about 5-6 compute-hour + 192GB main memory]: To run the entire evaluation directly from raw pcaps, it suffices to run the evaluation program with the parameters used in our evaluation.*
Preparation: Follow the previous instructions on running the example tests to setup the system.
Execution: To recompute all data, run the command:

```
python evaluate.py -recompute
```

In our experiments, it took about 90min to build the trees and about 5h to run BurstShark over all the raw pcap files. The resulting files are saved in `results/` folder and should be identical to the files[6] used in our evaluation.
Results: The claims are supported by reproducing the results as shown in the paper, see `results/` folder.
Use the provided plotting code to regenerate the paper's result figures (Figure 6, 7, and 8).
For Figure 6 of the paper, we record our information statistics for every seconds between 1 and 600 with $\theta = 2.2$; to reproduce the figure, run:

```
python plot.py 6
```

For Figure 7, the file `unknown-scores.csv` is used to plot the distribution of maximum scores over the unknown dataset; to reproduce the figure, run:

```
python plot.py 7
```

To reproduce Figure 8, one simply has to generate the result files for every θ between 1 and 5 by increment of 0.05, and the figure is generated by running:

```
python evaluate.py --all-theta
python plot.py 8
```

A.5 Notes on Reusability

We welcome any re-use of our artifact in further research.

Licensing Information Our Zenodo deposit contains both source code and datasets, each with separate licensing:

- **Source Code**: BSD 3-Clause License.
- **Datasets**: Creative Commons Attribution 4.0 International.

By using our artifact, you agree to comply with these respective licenses.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

[6] All result files are provided for convenience in the folder `precomputed-results/` in https://zenodo.org/records/15020614/files/plot-and-results.tar.gz.



USENIX Security '25 Artifact Appendix: Bots can Snoop: Uncovering and Mitigating Privacy Risks of Bots in Group Chats

Kai-Hsiang Chou*
National Taiwan University

Yi-Min Lin*
National Taiwan University

Yi-An Wang
National Taiwan University

Jonathan Weiping Li
National Taiwan University

Tiffany Hyun-Jin Kim
HRL Laboratories

Hsu-Chun Hsiao†
National Taiwan University
Academia Sinica

A Artifact Appendix

A.1 Abstract

This artifact appendix includes the following components: (1) analysis scripts for the Pushshift Telegram dataset; (2) basic chatbot implementations for various messaging services; and (3) the prototype implementation of SnoopGuard, our proposed secure group messaging protocol. First, we provide the analysis scripts for the Pushshift Telegram dataset, as used in Section 3.2 of our paper. These scripts demonstrate the prevalence of repeatedly encountering chatbots across different groups. Our analysis reveals that when a user joins a new group containing chatbots, there is a 3.6% chance that at least one of the chatbots can recognize and associate the user with their previous interactions in other groups. Second, we include basic chatbot implementations for Discord, Keybase, LINE, Slack, and Telegram, along with tutorials on intercepting packets to and from these chatbots. These implementations help examine the information that chatbots can access and verify whether they support end-to-end encryption, as discussed in Section 4 of our paper. Finally, we provide a prototype implementation of SnoopGuard, our proposed secure group messaging protocol. This implementation is based on existing open-source libraries for the Signal Protocol and Message Layer Security (MLS). Additionally, we include benchmark scripts and the performance evaluation results referenced in Section 6.2.2 and Appendix D of our paper. Our prototype implementation shows that sending a message in a group of 50 users and 10 chatbots takes about 10 milliseconds when integrated with MLS.

A.2 Description & Requirements

Our artifacts include three components: (1) analysis scripts for the Pushshift Telegram dataset; (2) basic chatbot implementations for various messaging services; and (3) the prototype implementation of SnoopGuard, our proposed secure group messaging protocol.

*Both authors contributed equally to this research.
†Hsu-Chun Hsiao (hchsiao@csie.ntu.edu.tw) is the corresponding author.

A.2.1 Security, privacy, and ethical concerns

The risks of this artifact align with general expectations and do not pose significant threats to system security, data privacy, or ethical concerns.

A.2.2 How to access

The latest version of the source code is available on GitHub (https://github.com/csienslab/snoopguard-artifact). The specific version used in the main paper can be accessed on GitHub (https://github.com/csienslab/snoopguard-artifact/tree/5649ba9) or on Zenodo (https://zenodo.org/records/14729613).

A.2.3 Hardware dependencies

Pushshift Telegram Dataset Analysis. The data analysis process consists of two parts: (1) data preprocessing, and (2) data analysis. The first step involves downloading the Pushshift Telegram dataset and importing it into PostgreSQL, which requires approximately 1 TB of storage. To simplify reproduction, we have included the processed and extracted data in our source code as several pickle files (Python's object serialization format). These pickle files can be used directly without downloading the full dataset. The second step has no specific hardware requirements.

Basic Chatbot Implementations. No specific hardware requirements.

SnoopGuard Reference Implementation. The implementation requires at least 16GB of RAM to complete the benchmark. The benchmark tests were executed on a Mac mini with an Apple M2 processor and 16 GB of RAM. The provided

Geekbench script assumes an AArch64 (ARM64) architecture. Running the script on other architectures may require modifications to download compatible versions.

A.2.4 Software dependencies

Pushshift Telegram Dataset Analysis. We use PostgreSQL for data storage and extraction, and Python for writing the analysis scripts. As mentioned earlier, the processed and extracted data are included in our source code as several pickle files to facilitate reproducibility. These pickle files can be used directly, eliminating the need for PostgreSQL.

Basic Chatbot Implementations. Our chatbots are implemented on multiple platforms, including Discord, Keybase, LINE, Slack, and Telegram, using Node.js (v20.18.0) in our experiments. We use Wireshark to monitor HTTP and WebSocket connections and ngrok to create reverse proxies.

SnoopGuard Reference Implementation. Running the implementation requires a Go compiler (v1.21).[1] For resource-constrained benchmarks, Docker is also required.

A.2.5 Benchmarks

None.

A.3 Set-up

Pushshift Telegram Dataset Analysis. The analysis code can be found in the `analyze_pushshift` directory. To reproduce the data preprocessing process, download the Pushshift Telegram Dataset [2]. Set up a PostgreSQL database using the schema provided in `import_data/schema.sql`. Then, import the data using the scripts located in `import_data/*.py` from our repository.

Basic Chatbot Implementations. To execute the reference chatbot implementations, install Node.js, Wireshark, and ngrok. One also needs to apply for chatbot tokens from each platform's website. Detailed setup instructions are available in the README file of each implementation.

SnoopGuard Reference Implementation. Install the Go compiler.

A.3.1 Installation

Pushshift Telegram Dataset Analysis. To reproduce the data export process, run `analysis/dump_data_to_pickle.py` to generate the pickle files. Alternatively, one can use the pre-exported data provided in our repository to skip this step.

Basic Chatbot Implementations. The setup process for each chatbot platform varies. Detailed instructions can be found in the README file for each implementation.

SnoopGuard Reference Implementation. No additional steps are required beyond cloning the repository.

A.3.2 Basic Test

Pushshift Telegram Dataset Analysis. No functionality test is needed.

Basic Chatbot Implementations. Each chatbot is designed to echo any message it receives. To test this, send a message to the group. If the chatbot is set up correctly and receives the message, it will echo the input message.

SnoopGuard Reference Implementation. To test that the environment is set up correctly, run the test scripts using the following commands.:

```
go test ./pkg/user -v -timeout 0
go test ./pkg/chatbot -v -timeout 0
```

If the test case passes, the last line of the output would start with `ok`. All test cases should pass.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our analysis of the Pushshift Telegram dataset shows that 3.6% of users interact with the same chatbots across multiple groups. This result is demonstrated by experiment (E1) in Section 3.2.

(C2): Our analysis of chatbots on Discord, Keybase, LINE, Slack, and Telegram reveals that platform support for message access control and end-to-end encryption varies. Table 1 provides a detailed comparison. This finding is demonstrated by experiment (E2) in Section 4.1.

(C3): Our reference implementation of SnoopGuard achieves the benchmark results shown in Figures 4 and 5. Specifically, adding a chatbot to a group of 50 members takes about 2 milliseconds, regardless of the protocol. Sending a message to 50 members and 10 chatbots takes about 10 milliseconds when using MLS and about 5 milliseconds with the Sender Keys Protocol. These results are validated by experiment (E3) in Section 6.2.2 and Appendix D.

[1] At the time of writing, some of the packages we used do not support Go v.1.24 or higher.

[2] https://zenodo.org/records/3607497

A.4.2 Experiments

(E1): [Cross-Group Identification Analysis] *[30 human-minutes + 20 compute-hour + 1TB disk]*:
How to: Follow the instructions in Appendix A.3 and the README file to set up the Pushshift Telegram Dataset Analysis.
Preparation: As described in Appendix A.3 and the README file, data preprocessing involves downloading the Pushshift Telegram Dataset, importing it into PostgreSQL, and using the provided script to generate pickle files. Alternatively, one can use the preprocessed pickle files included in the source code to save time and reduce storage requirements.
Execution: Run `analysis/chatbot_analysis.py` to reproduce the results discussed in *Prevalence of Cross-Group Chatbots* (Section 3.2). Run `analysis/bot_user_encounter_ana.py` to reproduce the results from *User-Chatbot Encounters* (Section 3.2).
Results:

- **Prevalence of Cross-Group Chatbots:** The script prints the number of chatbots appearing in multiple groups. The results indicate that 253 chatbots (35.2%) were present in more than one group.
- **User-Chatbot Encounters:** The script `analysis/bot_user_encounter_ana.py` displays the results summarized in the main article at the end of the output. The analysis shows that 42,508 users (3.6%) encountered the same chatbot in multiple groups. Out of 4,155,927 user-chatbot pairs that interacted at least once, 97,813 pairs (2.4%) had multiple encounters.

(E2): [Chatbot Design Analysis] *[3 human-hour]*:
How to: Follow the instructions in Appendix A.3 and the accompanying README file to set up the basic chatbots on various platforms.
Preparation: The setup process for chatbots varies across platforms. Detailed instructions for configuring the chatbots and using Wireshark to intercept HTTP / WebSocket connections can be found in the corresponding README files.
Execution: Once the chatbots are set up, create a group on each platform and add the chatbot to the group. The steps for testing the chatbots' message access control are provided in Section 4.1.
Results: The expected results are shown in the rows *Message access control*, *Group E2EE (w/ bots)*, and *Hide sender* in Table 1. Respective evaluation criteria are listed below:

- Receiving chatbots' echo indicates that it has access to message.
- Plain text intercepted by Wireshark indicates absence of E2EE.
- User identifiers intercepted by Wireshark indicates not hiding sender.

(E3): [SnoopGuard Benchmark] *[6 compute-hour + 16GB RAM]*:
How to: Follow the instructions in the README file.
Preparation: Ensure that the Go compiler is installed.
Execution: Run the script `benchmark_roundtrip.sh` to perform a full benchmark. The execution times reported in the paper are the median values (`p50`).
Results: The output results will be stored in the `benchmark_results/native_roundtrip/`. In the result files, the keyword `IGA` indicates the use of SnoopGuard to achieve selective internal group anonymity (IGA). The keyword `Pseudo` indicates the use of SnoopGuard to achieve pseudonymity, as described in Section 5.5. If no keyword is present, the original protocol (Sender Keys Protocol or MLS) was used. The expected results are shown in Figure 5 and Figure 6.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Double-Edged Shield: On the Fingerprintability of Customized Ad Blockers

Saiid El Hajj Chehade[§], Ben Stock[‡], and Carmela Troncoso[§†]

§ EPFL, † Max-Planck Institute for Security and Privacy (MPI-SP) ‡ CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

In this work, we present three *scriptless* attacks to fingerprint ad-blocker configurations. We show that our attacks (1) identify 84% of the fitler-lists, (2) reduce the anonymity of *privacy-aware* users, in ad-blocker forum datasets, to a median of 48 users (0.2% of the population) using only 45 rules out of 577K, (3) capture fingerprints stable against constantly updated filter-lists, and (4) resist *SoTA* detectors and mitigations.

At a high level, the associated artifacts include

1. the *Main Fingerprinting Experiment [F]*, including fingerprint algorithms, forum dataset collection, our datasets, and evaluation and stats scripts. This component reproduces results in sections 5, 6, and 7.2 of the paper;
2. the *Web Measurement Study [W]*, including ad-blocker performance benchmarking and web-feature popularity measurement over the web. This component reproduces results in 7.1 and 7.3;
3. the *PoC Honey Page [H]*, an example page implementing a version of the attacks proposed. This component verifies the functionality of our attacks.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Executing experiments from the artifacts does not pose any security risk on the evaluators, as the code (1) only manipulates data associated with the experiments (included in the `./data` directory), and (2) sends GET requests to public services and APIs (namely, GitHub) at a reasonable rate, to prevent IP blocking. We also provide Docker containers to fully isolate the experimental setup. As for running the attacks in the Honey Page, they do not pose any security risk as they execute locally on the evaluator machine and only manipulate simple DOM elements and CSS.

Regarding ethical and privacy concerns, our experiments do not pose ethical or privacy risks. First, we scrape open-source (GPL-3) ad-blocker forum posts storing publicly-available information, which is standard practice in this line of work. On top of that, we do not store personal information of users and randomize their identifiers, to prevent privacy harms. Second, our web measurements implement precautions to not overload websites, like giving ample delay between visits (30 seconds). Our crawlers do not interact with the web pages beside the initial visit.

A.2.2 How to access

All components of the artifact including both code and datasets are accessible on zenodo.org/records/14736725. We also host the experiments' code (components [F] and [W]) on GitHub at spring-epfl/flfp. We also host the exemplary PoC honey pages (component [H]) on flfp-demo.github.io. The source code is available at flfp-demo/flfp-demo-builder.

A.2.3 Hardware dependencies

We initially ran the experiments on a Linux Ubuntu Server with 240 GB of RAM and 48 cores of Intel Xeon E5-2680. We verified the code's functionality on a MacBook M2 Pro with 32 GB RAM. However, on any desktop machine with fewer resources, running the fingerprinting algorithm on the full dataset size (18K users) will take considerably more time e.g., $\geq$ 16 hours.

A.2.4 Software dependencies

The main software required to run the artifacts is Docker (available at `https://docker.com`). Other libraries are bundled in the docker image we make available. All pieces of code can run on the Linux operating system with Python 3.10 or higher installed. The code has been tested on Ubuntu 20.04 LTS. The code relating to *main fingerprinting experiment* has also been tested on MacOS 14.2. We had several issues running code on Windows, so we recommend running the code on a Unix-based system.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Detailed instructions are available in the `INSTRUCTIONS.md` included with the artifacts. We provide step-by-step instructions to set up the environments here:
[F] Main Fingerprinting Experiment. The instructions are available in the repo's `README.md`. In summary,

1. Download the code from Zenodo or clone the GitHub repository, then `cd` into the `filterlist-fingerprint` directory.
2. Create a GitHub API key and add it to the `.env` file (a `.env.example` file is provided.)
3. Create `./data` directory or extract provided data.
4. Pull the Docker image
   `docker pull saiidhc/flfp:usenix`
5. Run the docker container using the command
   ```
   docker container create -it
   -env-file .env
   -v $(pwd)/data:/flfp/data
   saiidhc/flfp:usenix
   ```
6. Run experiments according to `docs/REPRODUCING.md`

[W] Web Measurement Study. After downloading the code from Zenodo or cloning the GitHub repository, then `cd` into the `measurements-adblockers` directory. Set up a Python virtual environment with the libraries in `requirements.txt`. After installing Docker, you can run the experiments directly with the commands provided in its `README.md`.
[H] PoC Honey Page. To start a local version of the honey page, make sure you have Python 3.8 or above and run the command `bash watch.sh` from the source code of the honey page repository.

In all future sections, commands mentioned regarding components [F] and [W] are relative to their respective root directories mentioned above.

A.3.2 Basic Test

[F] Main Fingerprinting Experiment. From within the docker container, run any command from `README.md` while restricting the size of the dataset. For example, you can download forum issues with `python scripts/run/issues.py forum=adguard pages_limit=1`. If successful, you should find the dataset collected in `data/issues/adguard/<timestamp>/issues_confs.csv`.
[W] Web Measurement Study. Inside the `performance/docker` folder of this codebase, run
`bash run.sh logs/../../websites_inner_pages-demo.json cpu 1 chrome`
If successful, you should find the measurements stored in `performance/docker/chrome/data`
[H] PoC Honey Page. After running `bash watch.sh`, the honey page should be available by visiting `http://localhost:8000`.

A.4 Evaluation workflow

Prior to running experiments, especially for *Main Fingerprinting Experiment*, we require scraping the user forum datasets and the filter lists with the commands provided in the repository and described in `docs/REPRODUCING.md`. This step can be susceptible to network-related failures or API rate limiting; so, to avoid these challenges, we provide pre-scraped user forum datasets and filter-list datasets in Zenodo's `data-usenix.zip`.

The claims and experiments are distributed over the artifact components as follows: [F] C1 to C5 and E1 to E5. [W] C6 and E6.

A.4.1 Major Claims

(C1): A substantial decrease in filter-rule coverage does not lead to a strong reduction in filter-list coverage, across all attacks. This is proven by experiment (E1) described in section 6.1 of the paper whose results are reported in table 3.
(C2.1): The attacks extract an entropy comparable to prior work (figuring in Table 2), between 0.44 and 0.75. The attacks uniquely identify between 14% and 20% of privacy-aware users in the datasets. This is proven by experiment (E2) described in section 6.2 of the paper whose results are reported in table 4.
(C2.2): The higher the degree of ad-blocker customization, the smaller the anonymity set of the privacy-conscious user is. This is also proven by experiment (E2) whose results are illustrated in figure 2.
(C3): The attacks can generate stable fingerprints that resist the frequent updates of filter lists by maintainers. The majority of rules usable for *general fingerprinting* remain valid through a period of 1,400 days. This is proven by experiment (E3) described in section 6.3 of the paper whose results are reported in the same section and illustrated in figure 3.
(C4): The adversary needs to control few domains (around 13) to identify almost all filter lists using *domain-specific* rules. This is proven by experiment (E4) described in section 6.4 of the paper whose results are reported in table 5.
(C5): Increasing the robustness of filter-lists to detection through forcing global rules can reduce entropy to below 0.2. The number of rules added is greater for AdGuard

than uBlock Origin (e.g., 442K and 317K respectively for CSS Animation attack) – which is manageable for uBlock Origin according to claim (C6). (C5) is proven by experiment (E5) described in section 7.2 of the paper whose results are illustrated in figure 5.

(C6): Forcing ad-blockers to use all rules, AdGuard users will suffer more from performance drawbacks – page load time and CPU performance – than uBlock Origin users. This is proven by experiment (E6) described in section 7.1 of the paper whose results are in figure 4.

A.4.2 Experiments

(E1): [Filter-list Coverage Study] [30 human-minutes + 3 compute-hour]: We evaluate how many filter-list and equivalence sets each attack and subset of rules can re-identify. The expected result is a table similar to table 3 containing the number/proportion of identifiable filter lists by each attack.
Preparation: From the repository base folder, navigate with `cd filterlist-fingerprint/`. Download filter lists and parse them with the commands in the `docs/REPRODUCING.md` Section (I), or use the pre-scraped filter lists from the ZIP. Similarly download or extract user issue dataset.
Execution: Run commands in `docs/REPRODUCING.md` sections II.1 and II.2. This will execute the fingerprinting algorithm on user datasets and filter-list datasets.
Results: Run command `python scripts/paper_stats/6_1_rule_and_list_coverage.py`. The generated tables will be outputed to the terminal and stored in `scripts/paper_stats/figures/rule_list_coverage_*_*.csv`.

(E2): [User Anonymity Study] [30 human-minutes + 6 compute-hour]: We evaluate the entropy and uniqueness of user fingerprints (targeted and general fingerprints) given our attacks and their filter-list coverage.
Preparation: Same as (E1).
Execution: Same as (E1).
Results: Run command `python scripts/paper_stats/6_2_reducing_user_anonymity.py`. The generated tables will be outputed to the terminal and stored in `scripts/paper_stats/figures/attack_stats.csv`.

(E3): [Fingerprint Stability] [30 human-minutes + 4 compute-hour]: We study how stable are rules that are essential for uniquely identifying lists. We sample the history of the filter list at various time points and identify rules that no longer exist by that time, to estimate rule life-time. We generate a time plot of the proportion of important rules that remain for the attacks over ad-blockers.
Preparation: Same as (E1) and download or extract the "github commits" data in `docs/REPRODUCING.md` Section (I).
Execution: Same as (E1) and Section II.3 from `docs/REPRODUCING.md`.
Results: Run command `python scripts/paper_stats/6_3_stability.py`. The generated figure will be outputed to the terminal and stored in `scripts/paper_stats/figures/rule_stability.csv`.

(E4): [Domain Coverage] [30 human-minutes + 4 compute-hour]: We iterate over domains appearing in domain-specific rules to maximize the number of lists newly identifiable if this domain is covered. We repeat it iteratively until either all lists are covered or no domain is useful.
Preparation: Same as (E1).
Execution: Sections II.1 and II.5 from `docs/REPRODUCING.md`.
Results: Run command `python scripts/paper_stats/6_4_domain_coverage.py`. The generated table will be outputed to the terminal and stored in `scripts/paper_stats/figures/domain_coverage.csv`.

(E5): [Iterative Robustness] [30 human-minutes + 1 compute-hour]: Simulate a game between adversary and maintainer. The maintainer enables global rules that the adversary uses to uniquely identify lists, in turn the adversary tries to find new rules to use.
Preparation: Same as (E1).
Execution: Sections II.1 and II.4 from `docs/REPRODUCING.md`.
Results: Run command `python scripts/paper_stats/6_4_iterative_robustness.py`. The generated figures will be outputed to the terminal and stored in `scripts/paper_stats/figures/iterative_robustness_*.csv`.

(E6): [Ad-blocker Performance Benchmarking] [20 human-minutes + 6 compute-hour]: Increasing the number of lists and measuring the CPU performance and load time impact on popular websites.
Preparation: From the repository base folder, navigate with `cd measurement-adblockers/`
Execution: Follow instructions in `README.md` > "Data Collection" > "Ad-blocker CPU Performance".
Results: Follow instructions in `README.md` > "Data Processing" > "Ad-blocker CPU Performance".

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Encrypted Access Logging for Online Accounts: Device Attributions without Device Tracking

Carolina Ortega Pérez* Cornell University
Alaa Daffalla* Cornell University
Thomas Ristenpart Cornell Tech

A Artifact Appendix

A.1 Abstract

Our artifact implements the protocol described in Section 4 of our paper. The artifact is composed of a `simulator.py` file, that can be used to run the end-to-end login and log retrieval algorithms from our protocol and a README explaining how to do so. Additionally, we have a `server2.py`, a `client2.py` and an `encryptor2.py` files, which represent the server, client, and encryptor components, respectively, of our protocol. At a high level, the server sends data related to the encrypted log of actions to the client, and the client sends requests to the encryptor to perform cryptographic operations. As mentioned throughout the paper, the protocol is agnostic to the authentication mechanism.

Moreover, our prototype includes files `encryptor_entry_size.py` and `algorithms_payload_size.py` that provide the size measurements (in bytes) for individual components, enough to analyze the payload of the remaining functions. We report the results from the prototype in Section 7 of our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our prototype fetches the serial number of the device on which it is running. However, the program runs locally and does not send this information anywhere or stores it persistently. Besides this, our prototype does not involve any human subjects or personal data. All account and authentication data is synthetic, for example we use synthetic randomly generated cookies, session identifiers, and user and relying party information.

A.2.2 How to access

Our artifact is publicly available in GitHub at https://github.com/alaadaff/csal_code and Zenodo at https://zenodo.org/records/14737179. .

*Authors contributed equally to this work.

A.2.3 Hardware dependencies

Our code is developed for and compiled on a MacOS device. We performed our experiments on a MacBook Pro with an M1 CPU and 8 GB of RAM running Sonoma v.14.6. We have not tested it on other operating systems or devices.

The only OS-dependent code is the function that retrieves the device's serial number, which utilizes macOS's `system_profiler`. This command does not require root privileges. The function is implemented in the `def getSerial()` function within the `encryptor2.py` file.

To adapt this code for other operating systems, replacing the function's implementation with an equivalent command should be sufficient. For example:

- On Windows, the serial number can be fetched using the `wmic bios get serialnumber` command.
- On Linux, the serial number can be obtained using the `dmidecode -s system-serial-number` command.

However, we note that both the Windows and Linux commands may require administrator or root privileges. We have not tested the code on these platforms.

A.2.4 Software dependencies

Our artifact is written in Python and requires installing the cryptography library version 42.0.5 and the pyhpke library version 0.5.3.

A.2.5 Benchmarks

None.

A.3 Set-up

To set up our prototype, all that is needed is downloading the code from Zenodo.

A.3.1 Installation

Our artifact can be downloaded from https://zenodo.org/records/14737179. It is implemented in Python and requires the cryptography and pyhpke libraries. These can be installed by running `pip install -r requirements.txt`.

A.3.2 Basic Test

Running `python3 simulator.py -e login-no-smuggle -i 1` should open two new terminals simulating a server and client respectively. It should run without any error, and print some performance values.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The payload for a login without key smuggling between the server and the client is 1,687 B. The response back to the client takes 3,224 B. The round-trip for one login takes 7ms (CPU time for both server and client to complete operations). This is reported in Section 7, in the login evaluation.

(C2): A log can be retrieved from the server and decrypted locally to display the log entries. The decrypted log contains the serial number of the device on which it is running. This is reported in Section 7, in the log retrieval evaluation.

(C3): A log retrieval payload from the server takes 2,482 B (no smuggling) and 2,631 B with smuggling. This is reported in Section 7, in the log retrieval evaluation.

(C4): A re-encryption request for a single session to be re-encrypted takes 2,653 B. This is reported in Section 7, in the re-encryption evaluation.

(C5): An entry in the encryptor database takes 2,263 B. This is reported in Section 7, in the login evaluation.

A.4.2 Experiments

(E1): *[login (no smuggle)] [approx. 1 human-minutes + 1 compute-minute]: Runs a login flow, sends data from the server to the client to the encryptor and back. This measures the login without key smuggling.*
How to: Run the command `python3 simulator.py -e login-no-smuggle -i 1`.
Results: The experiment opens two terminals, which print the CPU time and the payload size.

(E2): *[log retrieval] [approx. 1 human-minutes + 1 compute-minute]: Runs a log retrieval flow, sends data from the server to the client to the encryptor, where the log is decrypted and printed.*
How to: Run the command `python3 simulator.py -e history -i 2`.
Results: The experiment opens two terminals, which retrieves the log showing decrypted log entries.

(E3): *[log retrieval] [approx. 1 human-minutes + 1 compute-minute]: Computes the payload size for a single log retrieved from server.*
How to: Run the command `python3 log_retrieval2.py`.
Results: The experiment prints the size of a single log returned from the server during log retrieval.

(E4): *[other algorithm payloads] [approx. 1 human-minutes + 1 compute-minute]: Computes the payload size for re-encryption and action.*
How to: Run the command `python3 algorithms_payload_size.py`.
Results: The experiment prints the size of the payload at the different stages of the re-encryption and action.

(E5): *[encryptor storage] [approx. 1 human-minutes + 1 compute-minute]: Computes the size of the different components in an encryptor entry.*
How to: Run the command `python3 encryptor_entry_size.py`.
Results: The experiment prints the size of each element on each row of an encryptor entry.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: How to Compare Bandwidth Constrained Two-Party Secure Messaging Protocols: A Quest for A More Efficient and Secure Post-Quantum Protocol

Benedikt Auerbach
PQShield

Yevgeniy Dodis
New York University

Daniel Jost
New York University

Shuichi Katsumata
PQShield and AIST

Rolfe Schmidt
Signal Messenger

30 May 2025 v0.0

A Artifact Appendix

A.1 Abstract

This artifact contains the code used to implement and evaluate the post-quantum ratcheting protocols described in the paper. The paper presents protocols using three KEM variants - Ratcheting KEM (RKEM), Unidirectional KEM (UniKEM), and Bidirectional KEM (BiKEM) - and using two sampling strategies - Opportunistic and Non-opportunistic - for a total of six protocols. The primary goal of these post-quantum secure messaging protocols is to provide post-quantum post compromise security (PCS), and we quantify this by measuring the size of vulnerable message sets ($|VulM|$) that can be recovered by an adversary after compromising one of the parties.

While this metric is natural, it also depends fundamentally on the underlying messaging behavior or the protocol participants. Among other issues, when the underlying messaging behavior is random, the size of $|VulM|$ uponcompromise will be a random variable. Without specifying a messaging behavior, these protocols are notcomparable: we can find a messaging behavior where any one of the six protocols produces the smallest $|VulM|$. Thus we simulate five messaging behaviors that approximate realistic usage by users of Signal Messenger, modeling balanced and unbalanced communication, as well as the behaviors of primary devices (cell phones that are usually online) and linked devices (often desktops that are online only when the user is at their desk). All of these behaviors are implemented in this artifact.

Finally, this artifact provides a top-level program that will perform a simulation of any of the protocols under any of the messaging behaviors and output statistics about the distribution of sizes of the resulting $|VulM|$. Scripts are also provided torun all combinations of protocols and messaging behaviors, then use the output data to reproduce the charts presented in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is available on Zenodo at the following URL:

```
https://zenodo.org/records/15571277
```

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

This artifact requires the following software dependencies:

- `Rust 1.88.0`: On Unix-like systems, the rust toolchain can be installed with `rustup` using the command:

  ```
  curl --proto '=https' --tlsv1.2 \\
    -sSf https://sh.rustup.rs | sh
  ```

  To install the specific version, you can run:

  ```
  rustup install 1.88.0
  rustup default 1.88.0
  ```

  but be aware that this will override the default version of Rust on your system, affecting other Rust projects.

- `Gnuplot`: On Debian Linux, run the command:

  ```
  sudo apt install gnuplot
  ```

On MacOS run the command:

```
brew install gnuplot
```

A.2.5 Benchmarks

None.

A.3 Set-up

With the Rust toolchain and Gnuplot installed, the artifact can be built by running the following command in the root directory of the project:

```
cargo build --release
```

This will compile the Rust code and produce an executable in the `target/release` directory.

A.3.1 Installation

Once compiled, no further installation is needed. The executable can be run to give basic usage information with the following command:

```
./target/release/usenix2025sims --help
```

A.3.2 Basic Test

The program can be used to run simulations of different protocols under different messaging behaviors described in the paper by executing the following command:

```
./target/release/usenix2025sims  \\
   [OPTIONS] [PROTOCOL] [BEHAVIOR] \\
   [KEMTYPE] [OUTPUT] [CHUNKSIZE]
```

where:

- `[OPTIONS]`: Optional command-line options, such as `-numsteps` to specify the number of time steps for the simulation.
- `[PROTOCOL]`: The protocol to simulate, one of `opp-rkem`, `rkem`, `opp-uni-kem`, `opp-bi-kem`, `uni-kem`, `bi-kem`.
- `[BEHAVIOR]`: The messaging behavior, one of `balanced`, `pingpong`, `random`, `desktop-desktop-disjoint`, `primary-to-desktop`, `desktop-desktop-overlap`.
- `[KEMTYPE]`: The key encapsulation mechanism type, either `fs-rkem` to us a Forward Secret Ratcheting KEM or `non-fs-rkem` for the non Forward secret RKEM variant.
- `[OUTPUT]`: The data to output. Use `hist` to output a histogram of vulnerable message set sizes.
- `[CHUNKSIZE]`: The size of the chunks to process, e.g., `small` for 32B chunks, `medium` for 128B chunks, pr `large` for 512B chunks.

For example, to simulate the Opportunistic RKEM protocol with 32B chunks under the primary-primary messaging behavior for 100,000 time steps, run the following command:

```
./target/release/usenix2025sims \\
  --numsteps 100000 opp-rkem balanced \\
  fs-rkem hist small
```

You will then find two CSV files:

- `default_bal_opprkem_hist.csv` contains the frequency and cumulative distributions of the sizes of the vulnerable message sets upon compromize of parties A and B throughout the simulation.
- `default_bal_opprkem_stats.csv` contains the mean, standard deviation, and deciles of the vulnerable message sets exposed by parties A and B throughout the simulation.

A.4 Evaluation workflow

Scripts are provided to reproduce all experiments and charts in the paper.

A.4.1 Major Claims

The following claims are made in the paper, and are supported by the experiments in this artifact:

(C1): As SM protocols supporting larger bandwidth allow larger chunks to be sent, $|VulM|$ becomes smaller. However, since the protocol cannot become secure while one party is offline, the benefit of allowing larger chunks diminishes as the overlap of the parties' online time becomes small.

(C2): The security of an SM protocol hinges on the most insecure party; if one party is compromised, then all messages sent during that time become vulnerable, regardless of the peer's corruption state. As such, unbalanced communications amplify the time it takes to recover from a compromise, making the average and variance of $|VulM|$ larger.

(C3): A compromised party that is offline cannot recover, and thus, the average and variance of $|VulM|$ increase. The primary-primary and overlapping desktop-desktop model produce similar cumulative probability of $|VulM|$ as both send a similar number of messages and are online at the same time enough to avoid blocking; For primary-desktop we see a higher average of $|VulM|$ because the

communication is unbalanced and the online party will continue sending vulnerable messages while the other party is offline.

(C4): Opportunistic sending is consistently better (i.e., small average and variance of $|VulM|$) for small chunks, suggesting that wasting bandwidth is more harmful than sampling key material in advance. However, when the chunks become larger the benefit starts to diminish as they behave similarly. This is because as long as the party is offline, no recovery can happen — even if the online party finishes sending all information required to proceed to the next epoch secret.

(C5): Among the opportunistic protocols, Opp-RKEM and Opp-UniKEM perform similarly, and outperform Opp-BiKEM for all messaging behaviors except the "disjoint desktop-desktop" model.

A.4.2 Experiments

We provide a script that simulates all analyzed protocols undr all analyzed messaging behaviors, and produces the charts in the paper. These charts provide views of the generated data and support the claims above.

(E1): *[Main charts] [1 human-minute + 1 compute-minute + 4MB disk]: execute all protocols for all messaging behaviors and produce the charts in the paper.*
Execution: From the main directory, run the following commands:

```
cd charts/
./run\_tests.sh
```

Results: The figures from the main body of the paper will be generated in the directory `charts/figs/`. All charts generated by this script should match those in the paper exactly.

- To assess claim C1, the figure `fig4_primaryprimary_unikem_varchunksize.png` clearly shows that for the "primary-primary" messaging behavior which simulates two online cellphones, increasing the bandwidth limit and using larger message chunks leads to a clear reduction in the size of $|VulM|$. The figure `fig4_disjoint_dd_uniikem_varchunksize.png`, on the other hand shows the vulnerable message set size distributions of the same protocols, but under the"disjoint desktop-desktop" messaging behavior which simulates two Signal linked desktop devices that are never online at the same time. In this case, the advantages of sending larger messages disappear. These results are for the UniKEM protocol, but similar results can be seen for the other protocols.
- To assess claim C2, consider figures `fig5_unbal_uni_32.png` and `fig5b_unbal_rkem_32.png` which show the distributions of $|VulM|$ under balanced and unbalanced communication for the Opp-UniKEM and Opp-RKEM protocols respectively. For unbalanced communication we simulate situations where party A sends 9 times more messages than party B and vice versa. The figures show that the average and variance of $|VulM|$ are much larger for unbalanced communication. While not part of claim C2, the figures also bring out the inherent asymmetry of the Opp-UniKEM protocol, where security is more sensitive to a slowdown by party A than party B. This is because this protocol requires party A to transmit more data.
- To assess claim C3, consider figure `fig6a_online_overlap_katana_32.png`. Messaging behaviors that have both parties often online at the same time - the "primary-primary" and ""overlap desktop-desktop" models - yield similar cumulative probability distributions. The "primary-desktop" model, on the other hand, shows a higher average of $|VulM|$ because the online primary device is often sending messages to an offline desktop. The "disjoint desktop-desktop" model is even more extreme, as it simulates two parties that are never online at the same time, slowing PCS healing further.
- To assess claim C4, consider figure `fig6b_opp_vs_nonopp_unikem.png` which shows cumulative probability distributions of $|VulM|$ for the UniKEM and Opp-UniKEM protocol with different bandwidth limits. In the lowest bandwidth setting - a 32 byte limit per message - the opportunistic version of the protocol is clearly superior. This advantage persists but shrinks at the 128 byte limit, and disappears at the 512 byte limit.
- To assess claim C5, consider figure `fig7_vulnset_distributions.png` which plots cumulative probability distributions for all three opportunistic protocols under the four main messaging behaviors. It provides separate plots for compromise of user A and user B, since the opportunistic protocols have varying degrees of asymmetry. As claimed, the Opp-RKEM and Opp-UniKEM protocols perform similarly, and both outperform the Opp-BiKEM protocol for all messaging behaviors except the "disjoint desktop-desktop" model.

(E2): *[Appendix charts] [1 human-minute + 1 compute-minute + 6MB disk]: execute all protocols for all messaging behaviors and produce the charts in the paper and produce the charts in the appendix.*

Execution: From the main directory, run the following commands:

```
cd charts/
./appendix_tests.sh
```

Results: The figures from the appendix will be generated in the directory `charts/appendix_charts/`.

A.5 Notes on Reusability

While the scripts provided with this artifact are tailored to reproduce the charts in the paper, the underlying simulation code is designed for further experimentation. It is designed so that it is simple for a developer to implement and evaluate a new protocol or to add new messaging behaviors for evaluation of existing protocols - or both.

Adding a new protocol: To add a new protocol, implement the `MessagingScka` trait and the `ChunkedCKAVulnerability` trait for your protocol. Optionally, you can just implement the `Scka` trait for your protocol and use the `RkemMessagingScka` trait to get an implementation of the `MessagingScka` trait.

Adding a new messaging behavior: To add a new messaging behavior, implement the `MessagingBehavior` trait, which has a single function that returns a list of commands to be executed by the protocol participants at each time step.

Running simulations with a new protocol or messaging behavior: Once a developer has implemented a new protocol or messaging behavior, a simulation can be run as follows:

```
type Cka = MyCkaProtocol;
type MessagingBehavior = MyMessagingBehavior;
let mut mb = MyMessagingBehavior::new(...);
let numsteps = 100000usize;
let hist = orchestrator
    ::controlled_messaging_healing_test::<
    Cka,
    MessagingBehavior,
>(&mut mb, numsteps)
```

Looking through the code in `orchestrator.rs` will show other tests and measurements that can be performed, many of which are useful for testing the correctness of a protocol implementation. Look at the unit tests for existing protocols for examples.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: S/MINE: Collecting and Analyzing S/MIME Certificates at Scale

Gurur Öndarö[1], Jonas Kaspereit[1], Samson Umezulike[2], Christoph Saatjohann[1], Fabian Ising[2], and Sebastian Schinzel[1,2]

[1]*Münster University of Applied Sciences*
[2]*Fraunhofer SIT and National Research Center for Applied Cybersecurity ATHENE*

A Artifact Appendix

A.1 Abstract

We present the first comprehensive analysis of real-world S/MIME certificates used for digitally signing and encrypting emails. Our study focuses on the structure of the S/MIME PKI, the cryptographic properties of issued certificates, and their compliance with emerging standards such as the CA/Browser Forum's S/MIME Baseline Requirements. To conduct this study, we collected over 41 million unique X.509 certificates from public address books, i.e., LDAP servers, of which 38 million are suitable for use with S/MIME in email clients. Despite the inherent complexity of S/MIME trust chains, we developed and applied a custom verification tool that successfully reconstructed and validated certificate chains.

The artifact includes several components to facilitate reproducibility and support future research. It provides our custom-built chain verification tool, the scripts used for certificate collection and analysis, as well as the data required to replicate the core results presented in the paper. Due to the presence of personally identifiable information in the certificates (e.g., names and email addresses), we will provide the raw certificate dataset only for the second phase of the artifact evaluation but will not publish it. However, we do publish the list of IP addresses of the LDAP servers from which the certificates were collected, along with the crawling script, enabling others to replicate the data collection independently.

The goal of this artifact evaluation is to validate the replicability of our findings, evaluate the usability of our tooling, and confirm the completeness and transparency of our methodology. By releasing our tooling and methodology, we aim to enable the community to extend our work, reassess the ecosystem over time, and contribute to the ongoing improvement of S/MIME infrastructure security.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our scripts and tools will be executed on the virtual machine that we provide and host for the artifact evaluation. Our scripts do not contain destructive steps, and no security mechanisms have to be disabled. Please note that our public artifact contains only the scripts and tools, not the certificate dataset. Due to privacy concerns, we will not release the dataset publicly. However, it is made available to the artifact reviewers for their evaluation of the "Functional" and "Reproduced" badges.

A.2.2 How to access

Our artifact is available on Zenodo (`https://doi.org/10.5281/zenodo.15533203`) and GitHub (`https://github.com/FHMS-ITS/SMINE/`). It includes the scripts and tools we developed and used during our research. However, the certificate dataset will not be released due to privacy concerns, which we discuss in more detail in the paper. Consequently, some scripts cannot be executed without a connection to a MongoDB containing certificates.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

No specific requirements, any modern OS capable of running Docker containers and Python scripts will suffice. The Python requirements are bundled in a `requirements.txt` file as is customary.

For artifact evaluation, we provided reviewers with access to a local LDAP server containing certificates, which was used to assess the functionality of our LDAP crawler. Researchers can set up a similar local LDAP server serving certificates on their own machine to test our crawler.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To use the tools and scripts in the repository, you will need to have Python >=3.10 installed along with the required dependencies. We recommend using a virtual environment to manage dependencies.

```
apt install libsasl2-dev python-dev-is-python3
    libldap2-dev libssl-dev
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
export PYTHONPATH=$(pwd)
```

To test the crawler, set up a basic LDAP server, such as OpenLDAP (available via Docker at https://hub.docker.com/r/bitnami/openldap). Once the server is running, upload certificates to it by adding them to the `userCertificate;binary` attribute of the LDAP entries.

A.3.2 Basic Test

The script `~/SMINE/basic_checks.py` is provided to assert that all components are functional and correctly configured. It can be executed with Python as usual and will run a series of checks, printing instructions on how to proceed if any fail, and `All checks passed!` otherwise.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We implemented a crawler that crawls available certificates from an LDAP server using wildcard searches. (E1)

(C2): **Section 4** We collected 41,692,142 unique X.509 certificates from 2,879 LDAP servers, of which 38,374,783 fulfill the requirements for use with S/MIME in email clients. We described the skewed distribution of S/MIME certificates in Section 4.3, the Tables 2 and 3 show the distribution among countries and continents.
Section 5 We described the distribution of different configurations of Key Usage, Extended Key Usage and existing email address in Table 4. We found that 22,385,707 S/MIME certificates contained at least one email address, in total 22,672,795 email addresses and 806,746 unique domains. Of these, 15,865,134 contain at least one email address issued for a domain in the Tranco 1 million list[1], covering 30,872 domains. Using a list of free mail domains[2], we found that 9,123,368 certificates contain at least one free mail address and name the most common free-mail and non-free-mail domains. In Figure 3, we show that most certificates have a validity period of multiple years and three years is the most common value. In Figure 4, we show the development of different public key algorithms over time. We find that CAs have begun implementing the S/MIME Baseline Requirements but most trusted certificates are still non-compliant. We show that a large subset of certificates is issued by a small number of CAs in Figure 5 and Table 5.
Section 6 We construct full certificate chains for the majority of certificates and connect 20% to a widely trusted root CA. All these values can be reproduced with experiment (E2).

(C3): In Section 7 we analyzed the cryptographic keys contained and found vulnerabilities in 71,214 certificates. Table 7 shows an overview. The analysis can be reproduced with experiment (E3).

(C4): We implemented a tool that reconstructs and verifies S/MIME certificate chains for a given leaf certificate.

A.4.2 Experiments

(E0.1): (Optional) Execute the preprocessing scripts. These scripts were used to enrich the certificates in the database with additional information. This includes, for example, parsing X.509 certificates, classifying and flagging S/MIME certificates, reconstructing and validating certificate chains, and labeling certificates as CA certificates. The README file in the `processing` directory contains more information about the individual tasks.
Preparation: Activate the environment as described in appendices A.2.2 and A.3.
Execution: Execute the `run.py` script in the `processing` directory to run all tasks.

```
cat assets/example_cert.json | python run.
    py certs > results.json
cat assets/example_host.json | python run.
    py hosts
```

(E1): [5 human-minutes + 30 compute-minutes]: Apply the crawler to the LDAP server (possibly in Docker) to crawl for certificates.
Preparation: The crawler can be tested against the LDAP server set up in appendix A.3.1. For the artifact evaluation, we provided an OpenLDAP Docker instance on the virtual machine, which served a certificate sample from our dataset and listened on port 389. Activate the environment as described in appendices A.2.2 and A.3.

[1] https://tranco-list.eu/list/VQ2QN

[2] https://gist.github.com/okutbay/5b4974b70673dfdcc21c517632c1f984

Execution: Execute the crawler script in the `ldap_crawler` directory.

```
python crawler.py 127.0.0.1,636,389 crawls
```

First, the crawler attempts to connect to port 636 on localhost. If port 636 is unavailable, it falls back to port 389. The crawling results (certificates) and logs are then written to the `crawls` directory.

Results: The `crawls` directory will contain a results file located in the results subdirectory, which includes the certificates crawled for the specified IP address. This experiment demonstrates the functionality of our crawler.

(E2): [1 human-hour, including validating the results. Little to no compute time when using the cache.]: Execute the scripts in the `analysis` directory.

Preparation: Activate the environment as described in appendices A.2.2 and A.3.

Execution: Execute the scripts:

```
python analysis/general_stats.py
python analysis/certs_per_host.py
python analysis/crawled_hosts_per_month.py
python analysis/certs_distribution.py
python analysis/smime_client_support.py
python analysis/smime_email_addresses.py
python analysis/smime_validity.py
python analysis/key_usage.py
python analysis/smime_key_algorithms.py
python analysis/smime_cas.py
python analysis/smime_br.py
```

Each script first attempts to load the results from a cache file located in the assets directory, which is generated after each run. If this file does not exist, the script will establish a connection to the database to execute the query. You can use the `refresh` argument to force the query to be executed directly against the database like:

```
python analysis/smime_br.py refresh
```

Results: The scripts will output the numbers, tables, and figures used in the paper.

(E3): [1 human-hour, including validating the results. Little to no compute time when using the cache.]: Execute the scripts in the `analysis/key_analysis` directory.

Preparation: Activate the environment as described in appendices A.2.2 and A.3.

Execution: Execute the scripts:

```
python analysis/key_analysis/
    weak_key_certs_count.py
python analysis/key_analysis/fastgcd_status.
    py
python analysis/key_analysis/badkeys.py
python analysis/key_analysis/ec_keys.py
python analysis/key_analysis/
    factordb_status.py
python analysis/key_analysis/
    pwnedkeys_blocklist_merge_status.py
```

Results: The scripts would output the numbers, tables, and figures used in the paper, provided they are run with the appropriate dataset.

(E4): [5 human minutes]: Reconstruct certificate chains with the chain verification tool.

Preparation: Activate the environment as described in appendices A.2.2 and A.3.

Execution: Execute the Python tests in the `chain_verification` directory:

```
python tests/run_all_tests.py
```

To reconstruct a certificate chain using the chain verification tool, run experiment E0.1. This experiment includes the `cert_chains_task.py` script located in the tasks directory, which applies the chain verification tool to the given certificate. After execution, refer to the `results.json` file for the chain verification output.

Results: The chain verification tool provides a chain for the given certificate

A.5 Notes on Reusability

The crawler, in combination with the list of LDAP servers, can be used to recreate the certificate dataset. Please do so responsibly and follow best practices for scanning. Analysis scripts can be used and extended to analyze the resulting dataset.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: "Achilles: A Formal Framework of Leaking Secrets from Signature Schemes via Rowhammer"

Junkai Liang[1,*], Zhi Zhang[2,*], Xin Zhang[1,*], Qingni Shen[1,†], Yansong Gao[2], Xingliang Yuan[3], Haiyang Xue[4], Pengfei Wu[4], Zhonghai Wu[1,†]
[1]Peking University, [2]The University of Western Australia,
[3]The University of Melbourne, [4]Singapore Management University
{ljknjupku, zzhangphd}@gmail.com, zhangxin00@stu.pku.edu.cn,
qingnishen@pku.edu.cn, garrison.gao@uwa.edu.au, xingliang.yuan@unimelb.edu.au,
haiyangxc@gmail.com, pfwu@smu.edu.sg, wuzh@pku.edu.cn

A Artifact Appendix

A.1 Abstract

To demonstrate our main results in the paper, we prepare our artifact which contains 3 main components: 1) The attacker's code related to fault injection (in the directory 1.Fault-injection/). This code implements the specific fault injection techniques used in the research, including memory profiling, memory massage and online row refresh. This code needs to be run on a personal computer with vulnerable DIMMs. We have promised to share this component for availability. 2) The attacker's code that analyses the faulty signature (in the directory 2.Post-analysis/). This code analyses the faulty signatures and outputs the leakage of the secret bits. We have promised to share this component for availability and functionality. 3) The automation tool for Achilles which analyses the signature schemes (in the directory 3.Automation-tool/). This code automates Achilles analysis for signature schemes. We have promised to share this component for availability and functionality.

A.2 Description & Requirements

The fault injection part in our artifact has to be run on a machine with vulnerable DIMMs and other parts can be run on any general-purpose personal computer.

A.2.1 Security, privacy, and ethical concerns

Our artifact may be destructive. When running with vulnerable DIMMs, the display screen may stop working and the system may abort. We recommend doing a data or system backup before starting the experiment.

A.2.2 How to access

Stable URL pointing to the latest version of our artifact: https://doi.org/10.5281/zenodo.14735639.

Stable URL for the feedback and dynamic changes: https://github.com/liang-junkai/Achilles.

A.2.3 Hardware dependencies

The fault injection part in the artifact needs to be run on linux kernel in a machine with vulnerable DIMMs. If the underlying hardware is not vulnerable, the program will work but will not publish bit-flip information. We perform the experiments using a vulnerable Samsung DDR3-1300 4G DIMM (part number: M473B5273DH0-YK0) deployed in a Lenovo T420 equipped with Intel Core i5-2430M CPU (Sandy Bridge) in Ubuntu 20.04.

A.2.4 Software dependencies

To run the post analysis part in our artifact, the corresponding library, e.g., wolfssl, relic and liboqs should be installed first. In component 2, we evaluated 3 libraries which are `relic-toolkit-0.6.0`, `wolfssl-5.6.6`, and `liboqs-0.10.0` on ubuntu 20.04.

A.2.5 Benchmarks

None.

*: The authors contribute equally to this paper.

†: Corresponding author. This work was supported by the National Key R&D Program of China under Grant No. 2022YFB2703301, PKU-OCTA Laboratory for Blockchain and Privacy Computing and the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant.

A.3 Set-up

A.3.1 Installation

We recommend installing 3 libraries: WolfSSl-5.6.6, Relic-0.6.0, Liboqs-0.10.0.

A.3.2 Basic Test

After installation, test the executable file in each directory (using the ReadMe.md), to see if the library is correctly linked.

A.4 Evaluation workflow

A.4.1 Major Claims

The major claims related to our artifact are about the feasibility of the fault injection and the post-Rowhammer analysis, and the automation tool.

(C1): The fault injection component in our artifact contains all the code related to Rowhammer preprocessing and online injection, such as memory profiling, memory waylaying and double-sided hammering

(C2): The post-analysis component in our artifact can successfully recover the secret bits using faulty signatures.

(C3): The automation tool analyses the vulnerability of several signature schemes with correct input.

A.4.2 Experiments

Here are the instructions that we run each component.

(E1): *[Fault injection] [1 human-hour + 10+ compute-hour + 5GB disk]:*

How to: First run memory profiling code and identify vulnerable positions in the DIMMs. The perform the end-to-end Rowhammer attack and starts online injection.

Execution: In directory ./1.Fault-injection/memory-profiling, run the following codes for about 10 or more hours:

```
sudo ./rowhammer-sandy -d 1 -p 0.6
```

Then in directory ./1.Fault-injection/Online-injection, run the code in each subdirectory

```
sudo ./attack
```

If some offset page information is output, it means the test is successful. If the DIMMs are vulnerable and the result of the memory profiling is correct, faulty signatures will be output.

(E2): *[Post-analysis] [1 human-hour + 20 compute-minute]:*

How to: This component analyses the faulty signatures in libraries relic, wolfssl and liboqs.

Execution: In each subdirectory, first install each library following the commands in ReadMe.md, then run

```
./[scheme name]
```

Results: If the leakage information is output (position and value of the secrets), the test is successful.

(E3): *[Automation tool] [10 human-minutes + 5 compute-minutes]:*

How to: This code automatically analyses several signatures and output if they are vulnerable to our 2 attack algorithms, DFA and SCA.

Execution: In the directory, after installing the packages in the requirements.txt, run

```
python3 auto.py
```

Results: If the vulnerabilities of the schemes are output, the test is successfully.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Comprehensive Deniability Analysis of Signal Handshake Protocols: X3DH, PQXDH to Fully Post-Quantum with Deniable Ring Signatures

Shuichi Katsumata
PQShield & AIST

Guilhem Niot
PQShield, Univ Rennes, CNRS, IRISA

Ida Tucker
PQShield

Thom Wiggers
PQShield

A Artifact Appendix

A.1 Abstract

The Signal protocol relies on a handshake protocol, formerly X3DH and now PQXDH, to set up secure conversations. One of its privacy properties, of value to Signal, is deniability, allowing users to deny participation in communications. Building on Hashimoto et al.'s abstraction for Signal handshake protocols (USENIX'25), our paper proposes a unified framework to analyze the deniability properties of these protocols, including post-quantum variants.

We further analyze post-quantum alternatives like RingXKEM, whose deniability relies on ring signatures (RS). By introducing a novel metric inspired by differential privacy, we provide relaxed, pragmatic guarantees for deniability. We also use this metric to define deniability for RS, a relaxation of anonymity, allowing us to build an efficient RS from NIST-standardized Falcon (and MAYO), which is not anonymous, but is provably deniable.

We implemented in C our proposed ring signature constructions and provided benchmark results demonstrating their concrete efficiency. Our artifact provides these implementations.

A.2 Description & Requirements

This artifact contains the experimental implementations of the ring signatures proposed in our paper, based on the signature schemes Falcon and MAYO. In addition, it includes the implementations of the prior ring signature works Falafl, Calamari and Raptor for performance comparison. The implementations of previous works are extracted from public sources, with minor updates to incorporate our benchmark code.

Our implementations of Falcon-RS and MAYO-RS consist in extending the respective reference C implementation of each base scheme to support signatures for rings of two users, as proposed in our paper. They concretely allow to generate signatures for a ring of two users, that is given one user's secret key and another's public key. They also allow to verify ring signatures given two public keys. We incorporate our benchmark code in each folder to produce the benchmarks of Table 3.

This section lists all the information necessary to recreate the experimental setup used in our paper.

A.2.1 Security, privacy, and ethical concerns

There is no risk introduced by this artifact for evaluators. All the operations are performed locally, in a typical set-up.

A.2.2 How to access

Our artifact can be permanently accessed on Zenodo at DOI 10.5281/zenodo.15571694.

A.2.3 Hardware dependencies

Part of our artifact requires AVX2 instructions support (available on Intel and AMD CPUs). Specifically the ring signatures Calamari and Falafl in the folder *prevworks/Calamari-and-Falafl*.

A.2.4 Software dependencies

For ease of reproducibility, we provide a Dockerfile specification that will compile the code of our artifacts and produce binaries required for our experiments.

The set-up instructions in the following section assume the installation of *podman* to execute our Dockerfile. Note that one may also use their Docker installation (in which case they should replace the *podman* command with *docker* in the instructions provided in the following section).

A.2.5 Benchmarks

Our artifact is used to produce the benchmarks available in Table 3 in our paper of FalconRS and MAYORS. Additionally, the code of previous ring signatures was used to compare the

efficiency of our schemes to prior works in the introduction and at the end of Section 6.

Note that more detailed benchmarks for prior works are available in Table 8 of the full version of our paper[1].

A.3 Set-up

This section includes all the installation steps required to prepare the environment to be used for the evaluation of this artifact.

A.3.1 Installation

1. The evaluator should first install *podman* by following the instructions at https://podman.io/docs/installation for their OS. For instance on Ubuntu, this is done by running `sudo apt-get -y install podman`.

2. Then, the evaluator should download the artifact's code. This can be achieved in two ways:

   (a) By downloading the repository ZIP available at DOI 10.5281/zenodo.15571694, decompressing it, and opening a terminal in the decompressed folder.

   (b) Or using Git (needs to be installed through the OS package manager) and by running in a terminal

   ```
   git clone \
     https://github.com/GuilhemN/ringsign.git
   cd ringsign
   git checkout 7cb6de1
   ```

3. Finally, the evaluator should compile the binaries required for the evaluation using the provided Dockerfile. This is achieved by running the following commands

   ```
   podman build -t rs-c-code .
   podman create --name=rs-c-code-tmp \
     rs-c-code
   podman cp rs-c-code-tmp:/out ./out
   podman rm rs-c-code-tmp
   ```

   Alternatively, instead of copying the binaries out of the container, the evaluator can execute them directly within the container by running:

   ```
   podman build -t rs-c-code .
   ```

   In this case, all benchmark commands in the following sections should be of the form `podman run -rm rs-c-code /out/<binary>` instead of using directly `./out/<binary>`. For example, instead of `./out/calamari-falafl-test_rs_iso`, one would run `podman run -rm rs-c-code /out/calamari-falafl-test_rs_iso`.

A.3.2 Basic Test

The evaluator can perform a simple functionality test by running one of the binaries produced during set-up:

```
./out/calamari-falafl-test_rs_iso
```

After a few seconds, benchmarks should appear for the key generation, signing, and verification. You can simply check for the presence of "keygen cycles", "signing cycles" and "verification cycles".

A.4 Evaluation workflow

A.4.1 Major Claims

Our paper makes the following claim:

(C1): Our ring signatures FalconRS and MAYORS outperform publicly-available implementations of prior ring signatures – Raptor, Calamari and Falafl – by a factor 32 - 66× for signing, and 146 - 1025× for verification. This is proven by the experiments (E1), (E2), (E3), (E4), (E5) described at the end of section 6 of our paper and in section J.3 of our full version[2] whose benchmark results are reported in Table 3 of our paper and Table 8 of our full version.

A.4.2 Experiments

(E1): [Benchmark FalconRS] [10 human-minutes + 1 compute-second]: benchmark the performance of Falcon-512 and FalconRS.
Execution: The benchmark can be executed by running `./out/rs-falcon-speed`.
Results: The evaluator should only collect numbers in the row "512", corresponding to the performance of Falcon-512 and FalconRS. They should compare the cycle counts to the ones provided in Table 3 of our paper (output in cycles by the script, in Megacycles in the paper): Keygen corresponds to the column *kg*, normal Sign corresponds to the column *sd*, 2-ring Sign corresponds to the column *rs_sd*, normal Verify corresponds to the column *vv*, 2-ring Verify corresponds to the column *rs_vv*. It is expected that cycle counts will differ on a different platform, but ratios between two cycle counts should be similar to what we obtained.
For the reference, on our test machine and as reported in Table 3 of our paper, we measured 6.2 Megacycles for

[1] https://eprint.iacr.org/2025/1090.pdf

[2] https://eprint.iacr.org/2025/1090.pdf

the Keygen, 0.26 Megacycles for normal signing, 0.74 Megacycles for 2-ring signing, 0.02 Megacycles for normal verification, 0.04 Megacycles for 2-ring verification.

(E2): [Benchmark MAYORS] [10 human-minutes + 10 compute-second]: benchmark the performance of MAYO 1 and 2 and the corresponding MAYORS.

Execution: The benchmark for MAYO 1 and MAYORS based on MAYO 1 can be executed by running `./out/mayo_bench MAYO_1 500`. The benchmark for MAYO 2 and MAYORS based on MAYO 2 can be executed by running `./out/mayo_bench MAYO_2 500`.

Results: The evaluator should compare the results of the commands with the values in Table 3. The output of the first command is used to fill the row $MAYO_1$ in the Table 3 of our paper, the output of the second command is used to fill the row $MAYO_2$ in Table 3.

The two commands have an output following the same format. They list benchmark results for different functions, with the average cycles count for one execution. We define the correspondence between the name of the functions in the commands output, and the name of the columns in Table 3.

Keygen corresponds to the function *mayo_keypair*, normal Sign corresponds to the function *mayo_sign*, 2-ring Sign corresponds to the function *mayo_rs_sign*, normal Verify corresponds to the function *mayo_verify*, 2-ring Verify corresponds to the function *mayo_rs_verify*. It is expected that cycle counts will differ on a different platform, but ratios between two cycle counts should be similar to what we obtained.

For the reference, on our test machine and as reported in Table 3 of our paper, we measured the following. For MAYO 1, we have 0.24 Megacycles for the Keygen, 0.88 Megacycles for normal signing, 1.1 Megacycles for 2-ring signing, 0.17 Megacycles for normal verification, 0.28 Megacycles for 2-ring verification. For MAYO 2, we have 0.65 Megacycles for the Keygen, 1.1 Megacycles for normal signing, 1.5 Megacycles for 2-ring signing, 0.09 Megacycles for normal verification, 0.16 Megacycles for 2-ring verification.

(E3): [Benchmark Raptor] [10 human-minutes + 5 compute-second]: benchmark the ring-signature Raptor for 2 users and verify claim (C1) that it is outperformed by FalconRS and MAYORS.

Execution: The benchmark can be executed by running `./out/raptor`.

Results: The command outputs **kilo**cycle counts for the key generation (preceeded by "bench keygen"), 2-ring signing (preceeded by "bench sign"), and 2-ring verification (preceeded by "bench verify") of Raptor. The evaluator can compare the observed performance of signing and verification to the 2-ring signing and 2-ring verification of FalconRS and MAYORS obtained in (E1) and (E2) to verify the performance ratios of claim (C1).

For the reference, on our test machine and as reported in the full version of our paper, we measured 27.1 Megacycles for the Keygen, 5 Megacycles for 2-ring signing, 2 Megacycles for 2-ring verification.

(E4): [Benchmark Falafl] [10 human-minutes + 5 compute-second]: benchmark the ring-signature Falafl for 2 users and verify claim (C1) that it is outperformed by FalconRS and MAYORS.

Execution: The benchmark can be executed by running `./out/calamari-falafl-test_rs_lat`.

Results: The command outputs cycle counts for the key generation (preceeded by "keygen cycles"), 2-ring signing (preceeded by "signing cycles"), and 2-ring verification (preceeded by "verify cycles") of Falafl. The evaluator can compare the observed performance of signing and verification to the 2-ring signing and 2-ring verification of FalconRS and MAYORS obtained in (E1) and (E2) to verify the performance ratios of claim (C1).

For the reference, on our test machine and as reported in the full version of our paper, we measured 0.1 Megacycles for the Keygen, 163 Megacycles for 2-ring signing, 76 Megacycles for 2-ring verification.

(E5): [Benchmark Calamari] [10 human-minutes + 30 compute-second]: benchmark the ring-signature Calamari for 2 users and verify claim (C1) that it is outperformed by FalconRS and MAYORS.

Execution: The benchmark can be executed by running `./out/calamari-falafl-test_rs_iso`.

Results: The command outputs follows the same format as for Falafl. It includes the cycle counts for the key generation (preceeded by "keygen cycles"), 2-ring signing (preceeded by "signing cycles"), and 2-ring verification (preceeded by "verify cycles") of Falafl. The evaluator can compare the observed performance of signing and verification to the 2-ring signing and 2-ring verification of FalconRS and MAYORS obtained in (E1) and (E2) to verify the performance ratios of claim (C1).

For the reference, on our test machine and as reported in the full version of our paper, we measured 119.5 Megacycles for the Keygen, 46581 Megacycles for 2-ring signing, 41250 Megacycles for 2-ring verification.

A.5 Notes on Reusability

The code of our ring signatures FalconRS and MAYORS can be reused for research purposes. We included a README in each folder `RS-falcon` and `RS-MAYO` to detail compilation options, and location of code examples to generate keys and ring signatures, as well as verify these signatures in C.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodol-

ogy followed for the evaluation of this artifact can be found at
https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: SparSamp: Efficient Provably Secure Steganography Based on Sparse Sampling

Yaofei Wang, Weilong Pang

A Artifact Appendix

A.1 Abstract

This artifact provides the implementation of SparSamp, an efficient provably secure steganography method for generative models. It includes code for embedding/extracting messages in text generation tasks using large language models (e.g., GPT-2, Qwen-2.5, Llama-3) and supports evaluation metrics such as embedding speed, decoding accuracy, and embedding rate. The artifact reproduces the key results in the paper, including:

- *High embedding rate approaching information entropy*
- *High embedding speed with $O(1)$ time complexity per sampling step*
- *Preservation of original probability distributions*

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

SparSamp is a dual-use technology. Users must comply with ethical guidelines and local laws when applying it.

A.2.2 How to access

- ***Repository**: Available on Zenodo with DOI: 10.5281/zenodo.15025436*

A.2.3 Hardware dependencies

- ***CPU**: Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz*
- ***GPU**: RTX 4090 × 1*
- ***Memory**: 128GB*

A.2.4 Software dependencies

- ***Python**: 3.8.5*
- ***Libraries**:*
  - `torch==2.2.2`
  - `transformers==4.41.2`

A.2.5 Benchmarks

- ***Datasets**: IMDB text samples (first 3 sentences per sample)*
- ***Models**:https://huggingface.co/*
  - `GPT-2:` `openai-community/gpt2`
  - `Qwen-2.5:` `Qwen/Qwen2.5-3B-Instruct`
  - `Llama-3:` `meta-llama/Llama-3.1-8B-Instruct`

A.3 Set-up

A.3.1 Installation

1. *Download the repository.*
2. *Install dependencies.*
3. *Download models (e.g., GPT-2).*

A.3.2 Basic Test

*Here, we can test the embedding and extraction functions of **SparSamp** in an example without using GPU, as shown below:*

1. *Enter folder* `[Basic Test]`
2. *Run the default configuration in main.py: python main.py*
3. *Expected Output: As shown in Figure 1.*

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *SparSamp achieves 100% decoding accuracy with message segment lengths $l_m \leq 1023$ (Section 4.3, Table 2).*
(C2): *High embedding rate approaching information entropy(Section 4.3, 4.5, 4.7, Table 2, 4, 7).*
(C3): *High embedding speed(Section 4.3, 4.4, 4.7, Table 2, 3, 7).*

```
(base) XXX@mac Artifact_sparsamp % python3 main.py

Using device: cpu

=== Initialization Phase ===
Message length padded to 1792 bits
Number of message bits used: 1792
model_name:../sparsamp_test/gpt/
Failed to load model locally, load GPT-2 directly from Transformers library
Model loaded successfully

=== Encoding Phase ===
Generated steganographic text:  When it came to languages, VF Native is
trying to use VF Native by showing different ways it is implementing it. I
suppose that these suggestions are too short, and I suppose that in my
opinion here VF Native not only gives what is really needed but also allows
to form an executable program that is portable for permanent use in native
platforms (and if you need to run VF Native, there are also uses for laro!).

VF Native as design

VF Native is designed with the N project as a strategic starting point for

=== Save and Verification Phase ===
Steganographic text saved to: stega_text.txt
Whether there is Token Ambiguity: No

=== Decoding Phase ===

=== Evaluation Results ===
Experimental Settings - Model: ../sparsamp_test/gpt/, Top-p: 0.95, Message
Segment Length (l_m): 64
Embedded 512 bits message in the generated 112 tokens
Message decoding result: Success
ATST: 2.86e-02 s/token
SITR: 0.00
Generation Speed: 34.9 tokens/s
Embedding Rate: 4.57 bits/token
Utilization: 94.7%
Embedding Speed: 161.0 bits/s
Decoding Speed: 177.3 bits/s
```

Figure 1: Expected Output of the Basic Test

A.4.2 Experiments

Enter folder `[Experiments/src]` *for running batch data*

Download the pre-trained language model to your local machine in advance. Then, modify the `params_dict['model_path']` *parameter in the* `get_statistics` *function within* `get_statistics.py` *to match the actual storage path.*

(E1): *Decoding Accuracy Validation [10 human-minutes + 1 compute-hour + 20GB disk]: Verify 100% decoding accuracy for message segment lengths $l_m \leq 1023$. Expected results: Success for all cases (Table 2).*

Preparation: *Ensure the model and tokenizer are loaded correctly. Run Python and set different* `block_size` *values to verify the extraction rate.*

Execution: *Run the following commands:*

```
block_sizes = [2, 4, 8, 16, 32, 64, 128,
256, 512, 1023]
result_list = []
for block_size in block_sizes:
    cur_result = get_statistics('sparsamp','1.0',
    block_size,'gpt2')
    result_list.append(cur_result)
pd.Dataframe(result_list).to_excel(
"result.xlsx",index=False)
```

Results: *Check the* `correct_decoded_rate` *in the logs or* `result.xlsx`*. Expected:* `1` *for all cases.*

(E2,E3): *Embedding rate and embedding speed verification [20 human-minutes + 3 compute-hour + 20GB disk]: Expected results: Utilization $\geq$ 93% (Table 3, 4, 7).*

Preparation: *Run Python and set different* `top_p` *values to verify the embedding rate and embedding speed.*

Execution: *Run the following command:*

```
top_ps = [0.80, 0.95, 1.00]
result_list = []
for top_p in top_ps:
    cur_result = get_statistics('sparsamp',
    top_p,64,'gpt2')
    result_list.append(cur_result)
pd.Dataframe(result_list).to_excel(
"result.xlsx",index=False)
```

Results: *Compare the* `embedding_rate`*,* `utilization`*,* `Embedding_Speed`*,* `Decoding_Speed`*,* `ATST`*,* `Generation_Speed` *and* `SITR` *with Table 3, 4, 7.*

Note the problems that hardware may cause*: Since some computers may cause floating-point precision issues, the cumulative probabilities may not match exactly. We ignore this issue and use the stored SE values from encoding during the decoding.*

A.5 Notes on Reusability

- ***New Models****: To integrate other models (e.g., Llama-3), modify* `model_path` *in* `get_statistics.py`*.*
- ***Custom Messages****: Replace* `message_bits.txt` *with any binary file.*
- ***Multi-modal Support****: SparSamp can be integrated with any model that provides explicit probability distributions by simply replacing the sampling component with SparSamp's embedding function.*

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: A Framework for Designing Provably Secure Steganography

Guorui Liao, Jinshuai Yang, Weizhi Shao and Yongfeng Huang

Tsinghua University

A Artifact Appendix

A.1 Abstract

A.1.1 Artifact Description

The artifact includes the following main components:

1. **Python implementations of steganography schemes:**
   - Two uniform steganography schemes mentioned in the paper.
   - Three provably secure symmetric steganography schemes proposed within the framework introduced in the paper.
2. **Testing scripts:**
   - Enable readers to verify the correctness and security of the steganography schemes.
   - Validate other theoretical properties discussed in the paper, such as stability and the theoretical lower bound of steganographic capacity.
3. **Steganography scripts for text channels:**
   - Steganography scripts that operate on text channels, enabling the conversion of secret information into normal text using the provided schemes, ensuring that the generated text is indistinguishable from text naturally produced by large language models.
   - Verify the encoding and decoding correctness of text-based stegotext.
   - Provide related statistical metrics mentioned in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact implements steganography schemes and generally does not pose security or privacy risks. It is developed for academic research and is not intended for unauthorized or malicious applications, such as covert communication for illegal purposes. Users must adhere to ethical guidelines and comply with applicable laws. The text-based steganography implementation involves large language models, which may generate varied outputs based on their training data. While this artifact does not intentionally influence model outputs, users should be aware of potential biases or unintended behaviors in the generated text.

A.2.2 How to access

https://doi.org/10.5281/zenodo.14737116

A.2.3 Hardware dependencies

This artifact is designed to run on a CUDA-supported GPU for efficient execution by default. Specifically, a CUDA-compatible GPU with at least 32GB VRAM is required for running text-based steganography schemes using large language models (e.g., **Llama3-8B-Instruct**). However, smaller models can be run on GPUs with less VRAM, depending on the model size being used.

A.2.4 Software dependencies

Please ensure the following tools are available on your device:

- **Miniconda / Anaconda**: Used for managing dependencies and creating an isolated Python environment.
- **Jupyter Notebook**: Required for running verification tests of the steganography schemes. Ensure Jupyter Notebook is installed in your environment.
- **CUDA-supported GPU environment**: Necessary for executing steganography schemes efficiently. All provided scripts and code are designed to run on a CUDA-supported GPU.
- **Large Language Model**: Required for text-based steganography. Download from **Hugging Face**. We recommend **Llama3-8B-Instruct**, but you may use other models like **Qwen2-7B-Instruct**, etc.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

The following steps guide the installation of dependencies and the main artifact:

1. **Download the artifact:** The artifact is available as a ZIP archive on **Zenodo**. Download and extract it.
2. **Set up a Python environment:** It is recommended to use **Miniconda** or **Anaconda** for environment management. Create and activate a new environment with **Python 3.12**:

```
conda create -n fdpss python=3.12
conda activate fdpss
```

3. **Install required dependencies:** Install all necessary Python packages:

```
python -m pip install -r requirements.txt
```

4. **Download the large language model:** We recommend downloading the large language model used for text steganography (e.g., **Llama3-8B-Instruct**) to your local machine first and then using the local path to load the model.

After completing these steps, the artifact should be ready for evaluation.

A.3.2 Basic Test

To verify that the artifact is installed correctly, run the following command:

```
python TextStego.py --model <path_to_model>
```

When using a downloaded language model, replace <path_to_model> with its local path. If no errors are reported, the installation is successful.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The two uniform steganography schemes mentioned in the paper ensure **Security** and **Correctness**. Additionally, the Cyclic-shift Uniform Steganography scheme achieves near-optimal **Capacity**, satisfying the theoretical bound:

$$H(U(n)) \geq C(U(n)) > H(U(n)) - 0.0861.$$

These are proven by the experiment (E1) and described in Section 4.3 (**Security**, **Correctness**) and Section 5.1.1 (**Capacity**).

(C2): The three provably secure symmetric steganography schemes (Differential-based, Binary-based, and Stability-based) proposed in the paper ensure **Security** and **Correctness**. Additionally, the Stability-based scheme guarantees the ability to embed at least one bit given any distribution with a minimum entropy of at least 1 (**Stability**), and the Differential-based scheme achieves an **Capacity** that satisfies the theoretical bound:

$$H - \log_2(1 + H \cdot \ln 2) - 0.0861 \leq C \leq H.$$

These claims are validated through experiment (E2) and further detailed in Section 3.4 (**Correctness**), Section 3.5 (**Security**), Section 5.1 (**Capacity**), and Section 5.2 (**Stability**).

(C3): For implementing steganography on text channels, we provide a script (`TextStego.py`) that performs textual steganography, verifies encoding and decoding **Correctness**, and provides **statistical metrics** for a single sample. These experiments correspond to scenarios (A and B) described in Table 1 of Section 5 and demonstrate the relevant metrics for an individual sample. These claims are validated through experiment (E3).

See `Schemes_Verification(English).ipynb` for more details on the validation of C1 and C2.

A.4.2 Experiments

(E1): [less than 1 min]: To verify the uniform steganography schemes, open `Schemes_Verification(English).ipynb` and follow the steps in the section **Verification of Uniform Steganography**. Run the Jupyter Notebook blocks corresponding to:

- **Test 1.1: Security** validation – Ensures the sampled frequencies align with the given n-uniform distribution.
- **Test 1.2: Correctness** validation – Verifies that encoding and decoding produce consistent results.
- **Test 1.3: Capacity** validation – Tests whether the Cyclic-shift Uniform Steganography scheme satisfies the theoretical bound:

$$H(U(n)) \geq C(U(n)) > H(U(n)) - 0.0861.$$

The execution results, corresponding explanations, and expected outcomes are provided within the Jupyter Notebook itself.

(E2): [less than 30 mins]: To verify the provably secure symmetric steganography schemes, open `Schemes_Verification(English).ipynb` and follow the steps in the section **Verification of Provably**

Secure Symmetric Steganography. Run the Jupyter Notebook blocks corresponding to:

- **Test 2.1: Security** validation – Ensures that the frequencies of multiple steganographic sampling results align with the given distribution.
- **Test 2.2: Correctness** validation – Verifies that encoding and decoding produce consistent results.
- **Test 2.3: Stability** validation – Tests whether the Stability-based scheme always allows embedding at least one bit when the distribution has a minimum entropy of at least 1.
- **Test 2.4: Capacity** validation – Verifies whether the Differential-based scheme satisfies the theoretical bound:

$$H - \log_2(1 + H \cdot \ln 2) - 0.0861 \leq C \leq H.$$

The execution results, corresponding explanations, and expected outcomes are provided within the Jupyter Notebook itself.

(E3): [About 1 minute to run once under the default setting]:
Execution: Execute the following command:

```
python TextStego.py --model <path_to_model>
```

Results: The script performs encoding and decoding for text steganography and outputs relevant information. Specifically, it:

- Generates a secret key and initializes `PRG_Encode`.
- Uses the `Encode` algorithm with secret information from `bit_stream.txt` to produce stegotext.
- Displays the generated stegotext along with the embedded secret bits.
- Uses the same key to initialize `PRG_Decode` and apply the `Decode` algorithm to extract the embedded bits from the stegotext.
- Outputs verification results to check whether the extracted bits match the original secret bits.
- Provides statistical metrics per token, such as embedding capacity (`C`), entropy (`H`), `C/H`, the frequency of tokens with zero embedded bits (`NR`), and sampling time per token (`T`).

For more details, refer to the README under **Implementing Steganography on Text Channels → Execution and Output**.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: BLens: Contrastive Captioning of Binary Functions using Ensemble Embedding

Tristan Benoit*†¶, Yunru Wang*†‡, Moritz Dannehl†‡, Johannes Kinder†‡

† Ludwig-Maximilians-Universität München, Germany
‡ Munich Center for Machine Learning, Germany
¶ Bundeswehr University Munich, Germany

A Artifact Appendix

A.1 Abstract

The artifact provides the datasets, implementation, and evaluation framework for BLens, a novel function name prediction model for stripped binaries. BLens combines COMBO (**CO**ntrastive **M**ulti-modal **B**inary embedding **O**ptimizer) to capture spatial relationships within the binary code and LORD (**L**ikelihood **O**rdered **R**egressive **D**ecoder) to improve function name prediction precision. We conduct ablation studies to validate the effectiveness of both components and compare BLens with several state-of-the-art tools in cross-binary and cross-project settings. Additionally, we introduce a strict setting to further evaluate generalization. This appendix contains the necessary information for reproducing the results and verifying the functionality of BLens

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact does not disable security mechanisms or process sensitive data, so it poses no security or privacy risks.

A.2.2 How to access

The artifact package can be accessed via GitHub repository, which includes the code, evaluation logs, and supplementary materials for implementation details. The Zenodo record contains all datasets in addition to the items available on GitHub.

A.2.3 Hardware dependencies

The minimum hardware requirements to reproduce the training and evaluation are 30 GB of memory, a GPU with 80 GB of VRAM, and 200 GB of available storage space.

A.2.4 Software dependencies

We recommend using a Python virtual environment, which may require virtualenvwrapper. Enchant is needed for the tokenizer, and Rust is required for VarCLR calculations. Installation instructions are provided in the `INSTALL.md` file in the GitHub repository.

A.2.5 Benchmarks

Pre-computed embeddings, tokenizers, and logs are contained in the data folder from the Zenodo record.

A.3 Set-up

A.3.1 Installation

1) Clone the package from the GitHub repository.
2) Download and extract `data.tar.gz` from the Zenodo record.
3) Follow the instructions in the `INSTALL.md`.

A.3.2 Basic Test

Once the installation is completed, users can do the following tests to check if the tool is properly installed.
1) Test the training work flow.

```
$ workon blens
$ CUDA_VISIBLE_DEVICES=0 nohup python3 RunExp.py -data-
    ↪ dir=DATA-FOLDER --basic-test >
    ↪ basic_test_training.txt &
```

2) Test the evaluation work flow.

```
$ cd evaluation
$ python3 new_experiments_evaluator.py -data-dir=DATA-
    ↪ FOLDER -d=basic_test --cross-project >
    ↪ basic_test_evaluation.txt
```

If the installation is successful, 11 files will be generated in the `DATA-FOLDER/xp/basic_test` directory.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): BLens achieves higher metrics compared to state-of-the-art competitors in the cross-project and strict settings as well as in the easier cross-binary setting. This

is proven by the experiment (E1), whose results are reported in Table 3 and 5, and experiment (E2) reported in Tables 1, 3 and 5.

(C2): COMBO pre-training and the new LORD decoder are shown to be beneficial in terms of F_1 score. This is proven by experiments (E3), (E4), and (E5) reported in Tables 6 and 8 of the ablation subsection (6.4).

A.4.2 Experiments

Given the time constraints of the artifact evaluation, we have structured our evaluation accordingly.

In the paper, (C1) is evaluated by retraining competitors such as XFL, SymLM, and AsmDepictor. Addtionally, BLens is not only trained on the full dataset in both cross-binary and cross-project settings; for better comparison, two variants are further trained on subdatasets generated from SymLM and AsmDepictor preprocessing. In all these settings, we consider F_1 scores as well as RougeL, Bleu, and VarCLR scores.

In the artifact evaluation, we propose only training a new BLens model in the cross-project setting. Other experiments are provided in the form of logs containing precomputed predictions. Lastly, we primarily evaluate our experiments using the F_1 score, which is our central metric.

In the paper, (C2) is evaluated using five models. The first is the full BLens model, trained for 80 epochs, similar to each ablation model. The variations include: BLens without pre-training (BL-NP); BLens with only the pre-training phase; BLens with a simple decoder that uses teacher forcing while maintaining a threshold; and BLens with teacher forcing but without implementing the threshold (SIMPLE-T0).

In the artifact evaluation, we propose to only train new models for BL-NP, and SIMPLE-T0. Other experiments are provided in the form of logs containing precomputed predictions.

(E1): *[5 human-minutes + 96 compute-hours]* Experiment E1 is to validate Claim C1 regarding the cross-project and strict settings. We validate this claim by training a new model from scratch in the cross-project setting and measuring F_1 scores in the cross-project and strict settings.

Execution: Train the model for around 96 hours:

```
$ workon blens
$ CUDA_VISIBLE_DEVICES=0 nohup python3 RunExp.py -data-
    ↪ dir=DATA-FOLDER -d=new-cross-project -config=
    ↪ main.json --cross-project -pretrain -train -
    ↪ inferBest > c1-cross-project-training.txt &
```

Get metrics for this new model:

```
$ cd evaluation
$ python3 new_experiments_evaluator.py -data-dir=DATA-
    ↪ FOLDER -d=new-cross-project --cross-project >
    ↪ c1-new-cross-project.txt
```

```
Main results
Setting: cross-project
       Bleu     F1  Precision  Recall  RougeL  VarCLR
test  0.242   0.46      0.655   0.354   0.393   0.648

Setting: intermediate strict
       Bleu     F1  Precision  Recall  RougeL  VarCLR
test  0.196  0.404      0.604   0.304   0.307   0.599

Setting: strict
       Bleu     F1  Precision  Recall  RougeL  VarCLR
test  0.094  0.294      0.482   0.211   0.243   0.568
```

Figure 1: Example of `c1-new-cross-project.txt` showing the result of retraining BLens in the cross-project setting.

Results: Review the file `c1-new-cross-project.txt`. Compare the F_1 score of BLens in the cross-project setting in Table 3 from the paper, which is 0.46, with the F_1 score in the table titled `Setting: cross-project` produced by the script. Compare the F_1 score of BLens in the strict setting in Table 5 from the paper, which is 0.294, with the one produced by the script. A difference of around ± 0.02 is expected due to weight initialization.

(E2): *[5 human-minutes + 10 compute-minutes]* Experiment E2 is to validate Claim C1 regarding other models and the cross-binary setting. We validate this claim by measuring F_1 scores from precomputed predictions in logs.

Execution: Compute metrics based on logs:

```
$ cd evaluation
$ python3 evaluator_c1.py -data-dir=DATA-FOLDER > c1-
    ↪ summary.txt
```

Results: Read the output `c1-summary.txt`. Compare metrics scores in Tables 1, 3 and 5 from the paper to the corresponding metrics in tables produced by the script. There should be no difference because this summary is based on precomputed predictions in logs.

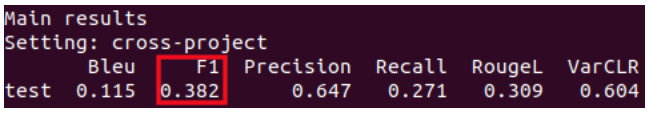
(E3): *[5 human-minutes + 54 compute-hours]* Experiment E3 is to validate (C2). We train a new BL-NP model from scratch and measure F_1 scores for this model.

Execution: Train the model for around 54 hours:

```
$ CUDA_VISIBLE_DEVICES=0 nohup python3 RunExp.py -data-
    ↪ dir=DATA-FOLDER -d=new-np -config=ablation-np.
    ↪ json --cross-project -pretrain -train -
    ↪ inferBest > c2-np-training.txt &
```

Get metrics for this new model:

```
$ cd evaluation
$ python3 new_experiments_evaluator.py -data-dir=DATA-
    ↪ FOLDER -d=new-np --cross-project > c2-new-np.
    ↪ txt
```



```
Main results
Setting: cross-project
       Bleu     F1  Precision  Recall  RougeL  VarCLR
test  0.115  0.382      0.647   0.271   0.309   0.604
```

Figure 2: Example of `c2-new-np.txt` showing the result of re-training BL-NP.

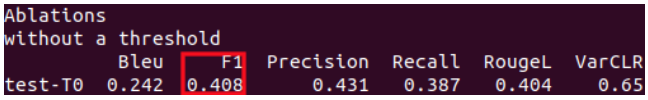
Results: Review the file `c2-new-np.txt`. Compare the F_1 score of BL-NP from Table 6 in the paper, which is 0.287, with the one found in the table titled `Setting : cross-project` generated by the script. Expect the score to be around 0.287 ± 0.10 due to the instability without COMBO pre-training phase. It should remain well below 0.445.

(E4): *[5 human-minutes + 32 compute-hours]* Experiment E4 is to validate (C2). We train a new SIMPLE-T0 model from scratch and measure F_1 scores for this new model.
Execution: Train the model for around 32 hours:

```
$ CUDA_VISIBLE_DEVICES=0 nohup python3 RunExp.py -data-
    ↪ dir=DATA-FOLDER -d=new-simple-t0 -config=
    ↪ ablation-simple.json  --cross-project -pretrain
    ↪  -train -inferT0 > c2-simple-t0-training.txt &
```

Get metrics for this new model:

```
$ cd evaluation
$ python3 new_experiments_evaluator.py -data-dir=DATA-
    ↪ FOLDER -d=new-simple-t0 --cross-project --
    ↪ evaluateNoTreshold > c2-new-simple-t0.txt
```



```
Ablations
without a threshold
          Bleu     F1  Precision  Recall  RougeL  VarCLR
test-T0  0.242  0.408      0.431   0.387   0.404    0.65
```

Figure 3: Example of `c2-new-simple-t0.txt` showing the result of retraining SIMPLE-T0.

Results: Review the file `c2-new-simple-t0.txt`. Compare the F_1 score of SIMPLE-T0 from Table 8 in the paper, which is 0.409, with the one found in the table titled `Setting: cross-project` generated by the script. Expect the score to be around 0.409 ± 0.01 due to weight initialization. Again, it should remain well below 0.445.

(E5): *[5 human-minutes + 5 compute-minutes]* Experiment E5 is to validate Claim C2 regarding other LORD ablation models. We validate this claim by measuring F_1 scores from precomputed predictions in logs.
Execution: Compute metrics based on logs:

```
$ cd evaluation
$ python3 evaluator_c2.py -data-dir=DATA-FOLDER > c2-
    ↪ summary.txt
```

Results: Read the output `c2-summary.txt`. Compare F_1 scores in Tables 8 from the paper to corresponding scores in the script-produced tables. No differences are expected since this is based on logs.

(E6): *Optional full re-training [5 human-minutes + 47 compute-days]* Experiment E6 is an optional experiment to re-train each BLens model and to measure their metrics. It requires around 47 days of computations with one GPU card.
Preparation: Modify line 3 of the `retrainAll.sh` script to set the correct `DATA-FOLDER` path.
Execution: Run the script:

```
$ nohup bash retrainAll.sh &
```

Results: Read outputs in the `evaluation` folder such as `cb-bl.txt` (for the BLens model in the cross-binary setting). Compare new tables with tables from the paper. Due to weight initialization, expect the F_1 score to differ by around ± 0.02 except in BL-NP case, in which it differs by ± 0.10.

A.5 Notes on Reusability

To employ a different dataset, you should begin by creating a dataset summary pickle file. This file should contain three lists categorizing functions into training, validation, and test sets, and each function within these lists should be represented by an array of the form [binary path, virtual address, real name, untangled name, placeholder, binary identifier, function identifier]. The binary and function identifiers are specifically required for DEXTER embeddings due to its precise database schema. Then, use `Tokenizers.py` to save a tokenizer tailored to your dataset. Subsequently, for each function, generate an embedding using the appropriate pre-processing tools provided by the creators of PALMTREE, DEXTER, and CLAP. These embeddings should be saved in dictionaries with keys as (binaryPath, virtual address) or (binary identifier, function identifier) for DEXTER embeddings, and values should be PyTorch tensors. Once these are saved using pickle, adjust the `RunExp.py` file to load these embeddings and the related dataset summary file and tokenizer.

To extend Blens with additional embeddings, you should modify the `COMBO.py` constructor to incorporate another module based on either `Dexter2Seq.py` for function embeddings or `PalmTree2Seq.py` for basic block embeddings. Update the `combo_input` function within `COMBO.py` to integrate this new module. Additionally, create a corresponding pickle dictionary with keys as (binary path, virtual address). Adjustments should be made to `RunExp.py`, `builder.py` and `COMBO.py` to load

new embeddings and to read the function encoder's new hyper-parameters added to the configuration file `configs/main.json`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: TRex: Practical Type Reconstruction for Binary Code

Jay Bosamiya[1], Maverick Woo[2], and Bryan Parno[2]

[1]*Microsoft Research*
[2]*Carnegie Mellon University*

A Artifact Appendix

A.1 Abstract

In the paper, we present TRex, a tool that performs automated deductive type reconstruction, using a new perspective that accounts for the inherent impossibility of recovering lost source types. Compared with Ghidra, a state-of-the-art decompiler used by practitioners, TRex shows a noticeable improvement in the quality of output types on 124 of 125 binaries.

This artifact contains the source code for TRex, scripts for producing reproducible benchmark binaries, as well as the scripts needed to run the evaluation presented in §5 of the main paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact poses no security, privacy, or ethical risks during execution. The code does not perform any destructive actions, nor does it disable any existing security mechanisms.

A.2.2 How to access

This artifact includes (i) the source code for TRex, (ii) reproducible scripts for the benchmarks, and (iii) evaluation framework to reproduce the results in §5 of the paper.

The artifact is archived at `https://doi.org/10.5281/zenodo.15611994`. Within this artifact, the TRex tool itself sits within `trex/`, the benchmark scripts sit within `trex-usenix25/benchmarks/`, and the rest of the files in `trex-usenix25/` are the evaluation scripts/framework.

The existence of two top-level folders reflects the fact that we have two GitHub repositories: `https://github.com/secure-foundations/trex` for the tool itself, and `https://github.com/secure-foundations/trex-usenix25` for the evaluation and benchmarks. The artifact contains a symlink from `trex-usenix25/trex` to `../trex` to establish the directory structure expected by our evaluation framework.

Details of software requirements, instructions, and more can be found in the top-level READMEs in each of the two top-level directories, but are also listed below.

A.2.3 Hardware dependencies

No special hardware is needed for running TRex, or indeed most of the evaluation, and a regular commodity x86-64 system suffices. While TRex itself works on ARM-based devices too, building the benchmark binaries requires an x86-64 machine. For future reference, we note that we ran our evaluation on a server with an Intel i9-10980XE (with 36 logical cores) and 256 GiB of system memory. We stress that the evaluation can be run using lower hardware specifications at the cost of a longer run time.

For replicating §5.3 in particular (i.e., evaluation against the Machine Learning-based prior work called ReSym), we recommend using a system with sufficiently powerful GPU (with CUDA support). The ReSym paper itself uses four A100 GPUs, thus we too used a server with four A100 GPUs.

Note that our evaluation scripts support running the GPU-intensive part of the computation on another server separate from the rest of the evaluation, if that is more convenient; details can be found in the relevant README.

A.2.4 Software dependencies

The evaluation has a small number of software dependencies that we list in the relevant READMEs. In short, we require an installation of Rust, Just, Python, uv, rename, and Ghidra. We have tested the evaluation locally on both Ubuntu 22.04 and 24.04, but recommend using the provided Dockerfile to ensure that all versions are set up correctly, and that paths and environment variables are correctly initialized.

A.2.5 Benchmarks

Benchmark binaries for the evaluation consist of COREUTILS and SPEC. For the former, no special access other than internet is needed, and the reproducible scripts will automatically download and build the binaries after confirming a

cryptographic checksum. For the latter, a license to SPEC CPU® 2006 is required to obtain the necessary source archive. Our reproducible scripts will confirm its cryptographic checksum and then use it for compiling the benchmark binaries.

A.3 Set-up

A.3.1 Installation

Within `trex-usenix25/.docker`, simply run '`make`' (without quotes). This uses `podman` to build an image, spin up a container, and drop you into a shell with all dependencies installed. The artifact itself will be at `/trex-usenix25/`; please ensure that the main TRex code is cloned or symlinked correctly at `/trex-usenix25/trex/` (running '`just trex`' within the `/trex-usenix25/` directory should correctly clone this if you don't have the folder).

For the benchmark binaries (outside the aforementioned container), run `make` within the directory of the benchmark of your choice, within `trex-usenix25/benchmarks`; this should automatically build all the benchmarks and place them into the relevant `evalfiles` folder.

A.3.2 Basic Test

Our runner system allows interactive configuration of jobs for a specific benchmark. In addition to COREUTILS and SPEC as benchmarks, we include a *basic-test* as an option. This is the default benchmark interactively. Thus, after spinning up the runner (i.e., executing '`just runner`' inside the `trex-usenix25/` directory, after installing all dependencies), simply select the defaults for everything (i.e., keep hitting ↩Enter) to run the full evaluation on the *basic-test* "benchmark". Note the '`just runner`' above is a command consisting of two words separated by a space.

This should finish in a few minutes, even if running only on a single core (although we recommend using multiple cores so that the various compilation steps are not slowing things down too much). Each runner sub-task (starting with "Confirm basic pre requisites" and finishing with "Summarize all metrics") will automatically report back with a success or failure. All jobs should show up with a checkmark indicating success. If any jobs fail, the runner gives you commands to run with more details and figure out what to fix.

The basic test runs all the same jobs as the full evaluation workflow using a couple of binaries instead of the complete benchmark. If the basic test succeeds, then the workflow on the complete benchmarks is expected to succeed.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): TRex qualitatively improves upon the decompilers used by practitioners (namely, Ghidra, Binary Ninja, and Hex-Rays / IDA Pro). This is demonstrated by Experiment E1, described in §5.1 of the paper.

(C2): TRex quantitatively outperforms Ghidra on real-world benchmarks (namely, COREUTILS and SPEC). This is demonstrated by Experiment E2, described in §5.2 of the paper.

(C3): TRex outperforms state-of-the-art Type Prediction (i.e., machine-learning based technique). This is demonstrated by Experiment E3, described in §5.3 of the paper.

A.4.2 Experiments

(E1): Qualitative comparison [<10 human-minutes + <10 compute-minutes].
Preparation: Make sure that the basic test described in Appendix A.3.2 is successful.
Execution: A quick approximation is to simply look at the `trex-usenix25/benchmarks/basic-test/evalfiles` directory after running the basic test described in Appendix A.3.2—the *easy* and *hard* linked list binaries and the corresponding results are named `test-linked-list-slot1.*` and `test-linked-list-slot2.*` respectively. Specifically, looking at the `*.trex-clike` and `*.trex-st` files are similar to Figure 5 of the paper. However, for presentation purposes in the paper, we use mildly different CLI flags that improve readability and understanding. Thus, for a more direct mapping to the paper, we recommend running (within the `trex-usenix25/trex/trex/` directory) '`cargo run -- from-ghidra tests/test-linked-list-slot2.{lifted,vars} -Zdisable-type-rounding -Zdisable-signed-integer-preference`'. Run a similar command but with `slot1` rather than `slot2` to obtain the "easier" variant described in the text. To obtain the decompiler outputs to compare against, simply load the relevant binary `*.ndbg-bin` into the decompiler to see the produced output types. To run the ablation described at the end of §5.1 of the paper, add the `-Zdisable-aggregate-type-analysis` argument to the `cargo run` command above. To see the expected warning (in the ablation for slot 1), add `-dd` to the command.
Results: The produced types at stdout (both structural and C types) should be virtually identical to Figure 5 in the paper. Similarly, the alternate "easier" variant (i.e., `slot1`) produces the expected types, and the ablation study produces the types described in the paper. Note that there may be minor naming or presentational differences, such as the use of curly braces for better readability in the paper, or numbering of types in the automatically-generated

type names—for example, `t31` in the paper might be α-converted to `t29` if there are minor differences in internal numbering, due to (say) debug logs.

(E2): Quantitative comparison [<10 human-minutes + up-to-overnight compute time].

Preparation: Make sure that the basic test described in Appendix A.3.2 is successful. Also, *outside* the container, within `trex-usenix25/benchmarks/<benchmark>`, run '`make`' to ensure that all the `.../evalfiles/*.binar.xz` (i.e., benchmark binaries) are generated (the run time depends on the parallelism afforded by your machine; while we found it to run on the order of minutes, it would not hurt to brew a cup of coffee/tea at this time). These binaries are what will be used by the runner for the evaluation.

Execution: Within `trex-usenix25/`, run '`just runner`', and select the `<benchmark>` when interactively prompted. Use the defaults for all other prompts, unless you explicitly want to change something (for example, use fewer than max cores). Even the jobs picked should be left at the defaults (see Experiment E3 for when the currently-not-selected ones are activated). At this point, the entire evaluation should run autonomously, and depending on the amount of parallelism, might finish in under an hour, or might take overnight. Some jobs (e.g., Ghidra) are known to be flaky, so the runner will automatically re-attempt them up to 3 times if it detects they have failed. Re-executing the runner with same settings will also re-use cached results for any previously-successful tasks. Thus, if the runner needs to be stopped halfway for some reason and then restarted, not much progress is lost.

Results: All results produced by the runner can be found in the `trex-usenix25/benchmarks/<benchmark>` directory. The `std-metrics.csv`, `summary.tex`, and `eval-*.pdf` files should match the results reported in the paper, specifically, Tables 2 and 3, and Figures 7 and 8. For the ReSym row in the tables, please see Experiment E3.

(E3): Comparison to ML-based technique [<30 human-minutes + ∼overnight compute time].

Preparation: Make sure to have successfully finished Experiment E2 first (the summarization scripts will overwrite previously-generated summary results; thus it might help to move the existing summary files to a separate directory for future reference, before moving on to this experiment). Then follow the instructions in `trex-usenix25/tools/evaluating_resym/README.md`, to set up a GPU-capable server with ReSym, and pass the correct environment variables.

Experiment: With the environment variables set up correctly, the runner will automatically recognize that it must evaluate ReSym too (i.e., the defaults automatically update such that the previously-disabled ReSym jobs get activated). The runner will also reuse any existing intermediate files from Experiment E2 whenever possible, so as to not waste time re-computing deterministic results.

Results: The generated summary files in the `.../<benchmark>` directory should match the results reported in the paper (Tables 2 and 3, and Figures 9 and 10), modulo randomization-induced noise due to the non-deterministic nature of the machine-learning based ReSym. Note: the "generous scoring" results are prefixed with `gen` in the produced results.

A.5 Notes on Reusability

The TRex tool should be independently reusable, even outside the evaluation. Instructions for this are provided in the TRex README. For customizing behaviors (e.g., to toggle the flags we describe in §3.2 of the paper), see TRex's `--help` output. Additionally, the benchmark scripts, and evaluation machinery should be usable for future research as-is. Introducing a new tool for future research to the evaluation scripts simply involves making sure that the tool can generate files of the `*.<toolname>-st` format, and adding a task to the runner to execute the tool.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Vest: Verified, Secure, High-Performance Parsing and Serialization for Rust

Yi Cai[†*] Pratap Singh[‡] Zhengyao Lin[‡] Jay Bosamiya[¶*]
Joshua Gancher[§*] Milijana Surbatovich[†] Bryan Parno[‡]

[¶]*Microsoft Research* [§]*Northeastern University* [‡]*Carnegie Mellon University*
[†]*University of Maryland, College Park*

A Artifact Appendix

A.1 Abstract

Paper: Many software vulnerabilities lurk in parsers and serializers, due to their need to be both high-performance and conformant with complex binary formats. To categorically eliminate these vulnerabilities, prior efforts have sought to deliver provable guarantees for parsing and serialization. Unfortunately, security, performance, and usability issues with these efforts mean that unverified parsers and serializers remain the status quo.

Hence, we present Vest, the first framework for high-performance, formally verified binary parsers and serializers that combines expressivity and ease of use with state-of-the-art correctness and security guarantees, including—for the first time—resistance to basic digital side-channel attacks. Most developers interact with Vest by defining their binary format in an expressive, RFC-like DSL. Vest then generates and automatically verifies high-performance parser and serializer implementations in Rust. This process relies on an extensible library of verified parser/serializer combinators we have developed, and that expert developers can use directly.

We evaluate Vest via three case studies: the Bitcoin block format, TLS 1.3 handshake messages, and the WebAssembly binary format. We show that Vest has executable performance on-par (or better) than hand-written, unverified parsers and serializers, and has *orders of magnitude* better verification performance relative to comparable prior work.

Artifact: This artifact contains the complete implementation of VestLib and VestDSL, as well as a suite of benchmarks on real-world binary formats including Bitcoin, TLS 1.3, and WebAssembly.

[*]Work done in part while at Carnegie Mellon University.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact is packaged as a Docker image, which contains the source code of VestLib and VestDSL, the dependencies (e.g., Rust and Verus), and the benchmarks for the three case studies. The binary data used in the benchmarks is publicly available and does not contain any sensitive information.

A.2.2 How to access

The artifact is available at:

https://doi.org/10.5281/zenodo.15611103

A.2.3 Hardware dependencies

To properly load the Docker image, we recommend a machine with at least 8 GB of RAM and 20 GB of disk space.

A.2.4 Software dependencies

While any operating system that supports Docker can be used to run the artifact, we recommend using a Linux-based OS (e.g., Ubuntu 20.04 or later) with Docker 27.5.1 or later installed.

A.2.5 Benchmarks

The artifact includes benchmarks for three real-world binary formats:

- Bitcoin 1K: 1,000 uniformly sampled blocks (out of ~870,000 blocks at the time of writing) from the Bitcoin main chain, with about 670 MB of data.
- TLS/Tranco: Handshake traces of TLS connections from making HTTPS requests the top 100 most visited domains according to the Tranco list [7].
- PolyBenchC [2]: A canonical Wasm benchmark [6] consisting of 30 C programs compiled to Wasm.

All benchmarks are already included in the Docker image, and can be run with the provided scripts (see subsection A.4).

A.3 Set-up

A.3.1 Installation

Download the Docker image from Zenodo (https://doi.org/10.5281/zenodo.15611103) and load it using the following command:

```
docker load -i vest-image.tar.gz
```

This will create a Docker image named `vest`.

> **Note:** The image file is create from macOS with the default BSD tar, so if you are using Linux, Windows, or Windows WSL, you might need to first extract the `tar.gz` with `tar -xvzf vest-image.tar.gz`, and then load the image with `docker load -i vest-image.tar`.

A.3.2 Basic Test

To start the Docker container, run the following command:

```
docker run -it --rm vest-image
```

This will start a new container based on the `vest` image.

> **Note:** The image is built for the linux/amd64 architecture, so if you are using an Apple Silicon (M1, M2, M3, etc.) machine, you might need to run the docker image with `docker run -platform=linux/amd64 -it -rm vest-image`.

From now on, every command should be run inside the Docker container.

Once inside the container, you can run the following command to verify that VestLib and VestDSL are correctly installed, respectively:

```
cd vest
make
verification results:: 477 verified, 0 errors
```

This will compile and verify VestLib.

```
cd vest-dsl
cargo run --release -- --help
Vest: A generator for formally verified
parsers/serializers in Verus

Usage: vest-dsl [OPTIONS] <VEST_FILE>
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Vest delivers runtime performance on-par with (or better than) state-of-the-art, hand-written, unverified parsers and serializers.

(C2): Vest achieves verification performance that is orders of magnitude better than prior work that also produces fast and verified parsers and serializers.

A.4.2 Experiments

(E1): [2 human-minutes + 10 compute-minute + 1GB disk]: This experiment is meant to support the claim **(C1)** above.

How to: The benchmark suites for Bitcoin, TLS 1.3, and WebAssembly are respectively located in the directories `vest-dsl/bitcoin`, `vest-dsl/tls`, and `vest-dsl/wasm`. Simply run `cargo bench` from within each of these directories to see the runtime performance comparison between Vest's parsers/serializers and the hand-written, unverified parsers/serializers (i.e., Rust Bitcoin [3], Rustls [4], and `wasmparser` [5]).

Results: The the Rust Criterion benchmarking library [1] that is used to run the benchmarks will output the results in a human-readable format, including the average time taken by each parser/serializer to process a single message, as well as the standard deviation of the measurements.

(E2): [1 human-minute + 5 compute-minutes + 1GB disk]: This experiment is meant to support the claim **(C2)** above.

How to: Similar to the previous experiment, the benchmark suites for Bitcoin, TLS 1.3, and WebAssembly are respectively located in the directories `vest-dsl/bitcoin`, `vest-dsl/tls`, and `vest-dsl/wasm`. Simply run `make` from within each of these directories to verify (and time the verification time) of the parsers/serializers generated by Vest.

Results: The verification results (number of verified components and errors, if any, as well as the total verification time) will be printed to the console. All produced parsers/serializers should verify within seconds without any errors.

All results of the experiments should be comparable to the results presented in the Evaluation section (Section 7) of the paper. We do not include the verification result for EverParse [8] in this artifact, as it will take several hours to verify.

A.5 Notes on Reusability

Vest is designed to be (re)usable by developers who want to get high-assurance and performant parsers and serializers

for their binary formats. To use Vest, most developers can define their binary format in VestDSL directly, whose syntax and semantics are defined in the paper. Expert developers can also use/extend VestLib directly to build custom parsers and serializers. Though extending VestLib requires a good understanding of the Verus verifier, using VestLib should be straightforward for developers familiar with Rust.

Vest is developed in an open-source repository, where you can find the detailed documentation, examples, and tutorials on how to use it.

https://github.com/secure-foundations/vest

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] Criterion.rs. https://github.com/bheisler/criterion.rs, 2024.

[2] Polybench/c. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1, 2024.

[3] Rust bitcoin. https://github.com/rust-bitcoin/rust-bitcoin, 2024.

[4] Rustls. https://github.com/rustls/rustls, 2024.

[5] wasmparser. https://github.com/bytecodealliance/wasmparser, 2024.

[6] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to speed with WebAssembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[7] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, Feb. 2019.

[8] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. dim Kobeissi, and J. Protzenko. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the USENIX Security Symposium*, Aug. 2019.



USENIX Security '25 Artifact Appendix
TYPEPULSE: Detecting Type Confusion Bugs in Rust Programs

Hung-Mao Chen, Xu He, Shu Wang, Xiaokuan Zhang, Kun Sun

A Artifact Appendix

A.1 Abstract

In this paper, we develop a static analysis tool called TYPEPULSE to detect three main categories of type confusion bugs in Rust including misalignment, inconsistent layout, and mismatched scope. We run TYPEPULSE on the top 3,000 Rust packages and uncover 71 new type confusion bugs, exceeding the total number of type confusion bugs reported in RUSTSEC over the past five years. We have received 32 confirmations from developers, along with one CVE ID and six RUSTSEC IDs. The artifact includes the source code of TYPEPULSE, dataset with GitHub issue links, and the detection results provided in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We take ethics seriously in this project. All Rust repositories we tested in the paper are publicly accessible on Github. During evaluations of our work, TYPEPULSE identified several previously unknown type confusion vulnerabilities in widely used software. In each case, we followed a responsible disclosure policy, and reported our discovered vulnerabilities to the developers. We also submitted our findings to the CVE program and the RustSec Advisory Database. We did not disclose those issues to anyone else. All the examples mentioned in the paper are the issues that have been acknowledged and fixed. The RustSec IDs issued are: RUSTSEC-2023-0046,RUSTSEC-2023-0047, RUSTSEC-2024-0408, RUSTSEC-2024-0424, RUSTSEC-2024-0426, RUSTSEC-2024-0431; The CVE ID is currently in the reserved status at the time of writing and will be released later.

A.2.2 How to access

The artifact can be downloaded from Zenodo: https://zenodo.org/records/14862485.

A.2.3 Hardware dependencies

We deploy TYPEPULSE on a server with 48-core Intel Xeon CPU ES-2630 and 256 GB memory. A regular PC can support our tool if the user doesn't run the detector with multiple threads.

A.2.4 Software dependencies

All the software dependencies can be installed automatically in the docker container. If setting up in local environment, users are required to install `rustc` with specific version `1.72.0-nightly` and the components including `rust-src`, `rustc-dev`, `llvm-tools-preview`, `miri`. The instructions of installing software dependencies are provided in the file `README.md`.

A.2.5 Benchmarks

Three datasets are included in our artifact:

First, 3000 Rust packages (`top3kscanned.txt`), which were downloaded before September 1, 2024.

Second, reports that include all TYPEPULSE detection results on 3000 packages (`report3k.csv`). The dataset includes all true positives, false positives, the corresponding function names, and the links of reported GitHub issues if true positives.

Third, existing RUSTSEC bugs for our evaluation (The directory of `rustsec`). The code of packages are provided in the directory by cases on RUSTSEC. In the subdirectories, we provide `README.md` to show all detection results. TYPEPULSE can also used to scan the packages to reproduce the detection results.

A.3 Set-up

A.3.1 Installation

We provide two ways to run TYPEPULSE:

First, execute dockerfile to automatically install and compile all the dependencies and TYPEPULSE into the container.

Second, to manually set up TYPEPULSE in the local environment, the user can also follow instructions in sections: *Requirements*, *First time to set up environment*, and *install with install.sh* in the file `README.md`.

A.3.2 Basic Test

The basic test can be started by setting up the docker we provide in artifacts. The instructions for running a simple functionality test are provided in the section: *Usage Tutorial: Run TypePulse in your package* of the file `README.md`. The instructions will provide the steps to download the package `candle-core`, which is included in Section 5.5, and reproduce the detection results.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We claim that TYPEPULSE can uncover 71 new type confusion bugs in Table 3 (while Table 3 only showed 50 bugs due to disclosure policies). This is proven by the experiment E1.

(C2): We claim that TYPEPULSE can achieve higher precision by enabling interprocedural analysis, as shown in Table 4. This is proven by the experiment E2.

(C3): We claim that TYPEPULSE can detect more type confusion bugs than Clippy and Rudra on existing RUSTSEC bugs, as shown in Table 5. This is proven by the experiment E3.

A.4.2 Experiments

Before running the following experiments, we need to ensure the required packages are downloaded. To download the source code of packages, we can run `crates_downloader.py` and `unarchive.sh`. We have detailed instructions in README.md.

(E1): 30 human-minutes + 1 compute-hour + 50GB disk
How to: First, verify **Preparation** has been done. Second, follow **Execution** to run the python script. Third, Compare the **Results** saved in `report.txt` to verify whether it matches the data of `report3k.csv`.
Preparation: Make sure the packages are all downloaded to the path `/home/crates/sources/`.
Execution: Use python3 to run the script `fast_typepulse.py`.
Results: The true positive and false positive detection results with interproceudural analysis from the packages in Table 3 will be saved in the `/home/crates/sources/[package-name]/report.txt`.

(E2): 30 human-minutes + 1 compute-hour + 50GB disk
How to: First, verify **Preparation** has been done. Second, follow **Execution** to run the python script. Third, Compare the **Results** saved in `report.txt` to verify whether it matches the data of `report3k.csv`.
Preparation: Make sure the packages are all downloaded to the path `/home/crates/sources/`.
Execution: Use python3 to run the script `fast_typepulse.py`.
Results: The true positive and false positive detection results without interproceudural analysis from the packages in Table 3 will be saved in the `/home/crates/sources/[package-name]/zreport.txt`.

(E3): 60 human-minutes + 2 compute-hour + 50GB disk
How to: The source code of the packages on existing RUSTSEC bugs have been included in the directory of `rustsec`. First, follow **Preparation** to install the tools as benchmarks,
Preparation: Install Clippy (version 0.1.72) and Rudra. Second, follow *Execution* to run TYPEPULSE and scan packages. Third, Compare the *Results* with corresponding `README.md` in `rustsec/rs-**`.
Execution: In each directory of `rustsec/rs-**`, run `cargo typepulse`.
Results: Detection results will be shown as standard output.

A.5 Notes on Reusability

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://zenodo.org/records/14862485.



USENIX Security '25 Artifact Appendix: From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification

A Artifact Appendix

A.1 Abstract

*Our artifact, **Lyso**, is a multi-target, multi-step directed greybox fuzzing (DGF) designed to verify static analysis results by leveraging semantic program flow information and alarm correlations. Lyso introduces novel seed selection, power scheduling, and step-tracking mechanisms to efficiently explore and confirm vulnerabilities flagged by static analysis tools. In our evaluation, Lyso achieved a 12.17× speedup over state-of-the-art fuzzers while confirming the highest number of true bugs and discovering eighteen new vulnerabilities.*

This artifact includes:

- ***Source Code:** Contains the implementation of Lyso, which is built on AFL v2.57b and uses LLVM 11.0.0.*
- ***Sequences of steps:** Sequences of steps derived from CodeQL, which guide the fuzzing process.*
- ***Evaluation Scripts:** Scripts to reproduce the experimental results on the Magma benchmark.*

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

- ***Execution Risks:** Lyso executes real-world applications with dynamically generated inputs. It is advised to run the artifact in an isolated virtual machine (VM) or a controlled testing environment to prevent potential security issues.*
- ***Crash-Induced Data Corruption:** As Lyso is designed to trigger crashes and anomalous behaviors, there is a risk of file corruption or unintended system behavior. Evaluators should ensure that no critical system files or sensitive data are accessible during execution.*
- ***Data Handling:** Lyso does not interact with personal or confidential data. However, test cases generated during fuzzing should be managed carefully to avoid exposing any unintended sensitive information.*
- ***Vulnerability Disclosure:** If evaluators identify new vulnerabilities using Lyso, they should follow responsible disclosure practices when reporting bugs to software maintainers.*

A.2.2 How to access

*Our artifact, **Lyso**, is publicly available and archived to ensure long-term accessibility. The artifact can be accessed at the following stable Zenodo reference: https://zenodo.org/records/14714504. Additionally, we will maintain a GitHub repository to facilitate continued development.*

A.2.3 Hardware dependencies

Lyso does not require any specialized hardware. It runs on general-purpose x86-64 architectures and has been evaluated on systems with Intel processors. There are no strict hardware dependencies such as GPUs, FPGAs, or specialized interconnects.

A.2.4 Software dependencies

Since all required software (e.g., compilers, fuzzing frameworks, Python packages) is handled through Lyso's Magma setup scripts inside the Docker container, evaluators do not need to install any additional dependencies manually and only need to ensure that they can run Magma on their system. To successfully launch Magma, we recommend running it on Ubuntu 22.04. Other Linux distributions may work, but they are not officially tested.

A.2.5 Benchmarks

*Lyso requires the **Magma** benchmark as the primary dataset for evaluating its fuzzing effectiveness. The Magma benchmark provides a diverse set of real-world programs containing known bugs, which serve as targets for Lyso's experiments. Buggy programs from Magma are included in the provided Docker environment and do not need to be manually downloaded or configured by the evaluator.*

A.3 Set-up

A.3.1 Installation

The installation of Lyso and its dependencies is fully automated using the Magma benchmark's Docker-based setup. Follow these steps to install and set up the artifact.

Step 1: Install Required Dependencies *Before using Magma and its scripts, install the necessary system dependencies:*

```
apt-get update && apt-get install -y \
  util-linux inotify-tools docker.io git
```

Step 2: Clone the Magma Repository *Download the Magma benchmark by cloning its repository:*

```
git clone https://github.com/HexHive/magma.git
```

Step 3: Move Lyso into Magma *To integrate Lyso with Magma, move the Lyso folder into the Magma fuzzers directory:*

```
mv /path/to/lyso magma/fuzzers/
```

This ensures that Lyso is correctly placed within the Magma environment.

A.3.2 Basic Test

To verify that Lyso is correctly installed and functional, perform the following steps. This test ensures that all required software components are in place and that Lyso can interact with the Magma benchmark.

Step 1: Using Captain Scripts for Fuzzing Campaigns *Magma provides the Captain scripts (located in `magma/tools/captain`) to build, start, and manage fuzzing campaigns. The script `magma/captain/run.sh` automates the process by:*

- *Building the required fuzzing images.*
- *Configuring parallel campaign execution.*
- *Managing fuzzing runs based on a configuration file (`captainrc`).*

Step 2: Configuring a Fuzzing Campaign *To run a 24-hour Lyso fuzzing campaign on a Magma target (e.g., `libpng`) one time and store the result in the workdir directory, update the `captainrc` file with the following settings:*

```
# WORKDIR: Directory for shared volumes
WORKDIR=./workdir

# REPEAT: Number of campaigns per program
REPEAT=1

# TIMEOUT: Campaign duration
# (supports s/m/h/d suffixes)
TIMEOUT=24h

# FUZZERS: List of fuzzers to evaluate
# (from magma/fuzzers/*)
FUZZERS=(lyso)

# Targets to fuzz with Lyso
# (from magma/targets/*)
lyso_TARGETS=(libpng)
```

Step 3: Running the Campaign *Once the `captainrc` file is configured, execute the run.sh script in the same directory.*

Step 4: Verifying the Output *During execution, the build and run logs of the campaign are stored in the workdir/log directory. In addition, the fuzzer logs and outputs can be found in workdir/ar/lyso/libpng/libpng_read_fuzzer/0/findings.*

To collect TTE and TTR data generated by Magma, refer to the monitor files located inside workdir/ar/lyso/libpng/libpng_read_fuzzer/0/monitor. Each monitor file is named using a timestamp (in seconds) representing the elapsed time since the start of the campaign. These files contain a CSV header and data rows that track the global campaign's bug reach and trigger counters at each timestamp.

If Lyso is functioning correctly, you should observe log entries indicating successful campaign execution and active fuzzer operations.

A.4 Evaluation workflow

A.4.1 Major Claims

The evaluation of Lyso aims to validate its effectiveness in verifying static analysis alarms. The major claims made in the paper are:

(C1): *The evaluation demonstrates that Lyso achieves an average 12.17× speedup over existing fuzzers in triggering bugs while also discovering the highest number of vulnerabilities in the Magma benchmark.*

A.4.2 Experiments

To validate Claim C1, we conducted experiments using Lyso on the Magma benchmark, comparing its performance

against state-of-the-art fuzzers, including AFL, AFL++, MOPT, AFLGo, Titan, FishFuzz, SelectFuzz, Parmesan. The primary objective is to measure Time-to-Exposure (TTE), Time-to-Reach (TTR), and Success Rate (SR) results.

(E1): *Performance Evaluation of Lyso [24 compute-hour per program]: This experiment evaluates Lyso's effectiveness in reducing Time-to-Exposure (TTE) and Time-to-Reach (TTR) while improving Success Rate (SR). The expected outcome is a significant decrease in TTE and TTR compared to eight state-of-the-art fuzzers, along with a higher SR in successfully triggering bugs.*

Preparation:

- *Set up the Magma benchmark following the A.3.1 instructions.*
- *Place Lyso in the magma/fuzzers directory.*
- *Ensure all baseline fuzzers are configured in the magma/fuzzers directory.*
- *Configure the* `captainrc` *file (located in magma/tools/captain) to define fuzzing parameters: FUZZERS=(lyso) TARGETS=(libpng libtiff libxml2 libsndfile lua poppler sqlite3 php openssl). This ensures that Lyso is executed on all buggy programs listed in Table 2.*

Execution: *Start the fuzzing campaigns for Lyso by executing the run.sh script located in magma/tools/captain. This will initiate the fuzzing process and generate a workdir directory, which will store all results, including TTE, TTR, SR, logs, and fuzzing statistics.*

Results: *To collect TTE, TTR and SR results, execute the following command: tools/benchd/exp2json.py workdir bugs.json*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: A Comprehensive Formal Security Analysis of OPC UA

Vincent Diemunsch
ANSSI & Université de Lorraine,
CNRS, Inria, LORIA, France

Lucca Hirschi
Université de Lorraine,
CNRS, Inria, LORIA, France

Steve Kremer
Université de Lorraine,
CNRS, Inria, LORIA, France

A Artifact Appendix

A.1 Abstract

Using the ProVerif cryptographic protocol verifier, we performed an in-depth formal security analysis of OPC UA, a standardized Industrial Control System (ICS) protocol. Our analysis uncovered attacks in several protocol configurations and established security proofs otherwise. In this artifact, we provide *(i)* the ProVerif models of the protocol and security properties, *(ii)* the scripts for reproducing the discovery of attacks and the security proofs (and the dependency graph between properties), and *(iii)* the vulnerability report used in the responsible disclosure.

A.2 Description & Requirements

Analyzing OPC UA with ProVerif for a given configuration. We provide the Python script `opcua.py` (configured by `config.py`) that instantiates our ProVerif model of OPC UA to a given configuration, and launches ProVerif on it. Table 1 depicts how configurations are defined by the presence or absence of the features and options and also specifies the corresponding script syntax.

Config. option	Admissible values	Script syntax
SecurityPol.	ECC | RSA	ECC | RSA
Mode	Enc | Sign | None	Encrypt | Sign | None
SessSec.	SSec | SNoAA	SSec | SNoAA
UserAuth.	Cert | Pwd | Ano	cert | pwd | anon
Leak	Ltk | Chk / Nok	lt_leaks | ch_leaks / no_leaks
Reopen	T | F	reopen / no_reopen
Switch	T | F	switch / no_switch

Table 1: Model configurations are defined by setting (possibly multiple) admissible values to each configuration option. "/" denotes mutually exclusive choices.

The script `opcua.py` relies on two Jinja2 template files: *(i)* `opcua-jinja.pv` that contains the protocol model and the queries corresponding to the security properties, and *(ii)* `config-jinja.pvl` that is used for configuring the model and ProVerif behavior. The script takes two arguments: the property to check and the configuration to consider. An example of a configuration is `ECC, Sign, reopen, SSec, pwd|cert, switch, no_leaks` (see Table 1). We provide details on script usage in the `README.md` file and in Appendix A.3.

Attack reconstruction. As explained in the companion report, ProVerif is able to automatically produce attack traces for all reported attacks. Providing `opcua.py` with the property and configuration for a given attack, ProVerif finds and reconstructs this attack. To ease this task, the script `reproduce_attacks.sh` automates those calls.

Exploring the lattice of configurations. As explained in the companion report, exploring all configurations by hand can be cumbersome and error-prone. We provide a lattice exploration tool `prove.py` to automatically explore the lattice of configurations and return the maximal configurations for which a given property holds.

Other artifacts. Some security properties rely on proving invariants in the first place. The dependencies between the properties and the invariants are provided in the file `dependencies.txt`. Note that the scripts `prove.py`, `prove.sh`, and `reproduce_proofs.py`, explained next, already take care of proving the conjunction of properties and invariants so that the end-user does not have to worry about those dependencies. We provide the dependencies for completeness. Finally, the vulnerability report is available in the file `vulnerabilities.pdf`.

A.2.1 Security, privacy, and ethical concerns

The attacks we found have been responsibly disclosed to the OPC Foundation, see `vulnerabilities.pdf`.

A.2.2 How to access

To retrieve the required files, `git clone` the following repository: `https://archive.softwareheritage.org/swh:1:rev:2f94c84f6125b7c07884369ba88e35a32032475b;origin=https://github.com/vdh-anssi/opc-ua_security`.

Note that this link has been updated with respect to the

paper to include improved scripts and documentation for reproducibility of the results.

A.2.3 Hardware dependencies

The attacks described in the paper, can be found and reconstructed with ProVerif on a standard laptop with 16 GB of RAM.

However, ProVerif requires more memory to establish the security proofs. To be able to establish all proofs, one needs 150 GB of RAM. Fast CPUs speed up such computations, but are not mandatory. Note that ProVerif is not multithreaded.

The lattice exploration tool is computationally much more demanding but is parallelized and thus benefits from multicore machines. We recommend a machine with at least 20 CPUs and 378 GB of RAM to run the lattice exploration tool for all properties.

A.2.4 Software dependencies

The results presented in the companion paper have been obtained with the development version of ProVerif, compiled from the sources on branch `improved_scope_lemma`. This is possible on Unix / MacOS / Windows platforms. Note that `Objective Caml` version 4.03 or higher is required. ProVerif can be installed from its sources available at https://gitlab.inria.fr/bblanche/proverif/. For using our scripts, you also need `Python` version 3.11 or higher with the template engine `Jinja2`.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The different alternatives to install an OCaml compiler are presented at https://ocaml.org/. Then, build ProVerif from its sources:

```
$ git clone https://gitlab.inria.fr/bblanche/proverif.git
$ cd proverif/proverif
$ git checkout 6a803aa13ccde13c0574912da93f029f86f63951
$ ./build
$ ./proverif --help
```

Make sure that the ProVerif executable is in your path and available from the command line.

Install `Python` version 3.11 or higher by following the instructions from https://www.python.org/downloads/. The template engine `Jinja2` (https://pypi.org/project/Jinja2/) can be installed with your preferred Python packet manager. For example using `pip3` (install with `sudo apt install python3-pip` on Unix platforms), one can install this dependency with the following command: `$ pip3 install jinja2`.

A.3.2 Basic Test

To check that the protocol model is indeed executable, one can query the reachability of events in a simple configuration. Note that reachability of an event returns `false`, whereas an unreachable event returns `true`, *i.e.*, the model is not able to execute a given event in the specified configuration. For example, executing: `$ python3 opcua.py -s -c "ECC, Sign, reopen, SSec, pwd|cert, switch, no_leaks"` will include (among others) the results:

```
- Query not event(S_open_channel(id_<xxx>,Sign,ECC,false))
is false.
- Query not event(S_open_channel(id_<xxx>,Encrypt,ECC,false))
is true.
```

as Sign mode is enabled, but not Enc (where `<xxx>` is an integer whose value may change among runs).

A.4 Evaluation workflow

A.4.1 Major Claims

We summarize here the claims of the companion paper, notably from Section 5 *"Analysis Results"*, that describes the attacks and the scope of the proofs.

(C1): *"Except for the signature oracle attack, all vulnerabilities we report were automatically discovered using ProVerif on our model."* See (E1).

(C2): *"Proofs for* $\mathsf{Conf_C}$ *and* $\mathsf{Conf_S}$ *are achieved with the maximal configuration for RSA; for ECC, we can prove the maximal configuration excluding None and SNoAA: (i) without Leak, as well as (ii) with Leak and either without Reopen, or without Switch. "*
"Proofs for $\mathsf{Conf_{Pwd}}$ *are for the maximal configuration without Leak. With Leak, we must choose between Reopen and Switch. We otherwise capture all configurations when considering in isolation SecurityPolicy and Mode."* See (E2).

(C3): *"Proofs for* $\mathsf{Agr_S^-}$ *can be obtained without Leak for Reopen + Switch; and with Leak without Reopen nor Switch."*
"Regarding $\mathsf{Agr_C^-}$*, one can additionally prove it for RSA + Leak + Reopen + Switch (without Enc)."* See (E3).

(C4): *"We developed a Python script to efficiently explore the verification of all configurations for a given property, and extract maximal configurations for which the property holds, minimal configurations for which the property does not hold, as well as minimal configurations for which ProVerif does not terminate successfully."* See (E4).

A.4.2 Experiments

(E1): [Attack finding and reconstruction] [5 human-minutes + 30 compute-minutes + 150 MB disk]:
How to: For each attack, the `README.md` file provides detailed instructions on how to automatically find the attack and produce an attack trace in a PDF file (or, if

applicable, inspect how ProVerif constructs a clause that contradicts the query).

Execution: Run `./reproduce_attacks.sh`, which calls `opcua.py` for each of the attacks.

Alternatively: run `opcua.py` manually. For instance, one can refind the attack *§5.1 Race Condition for User Contexts* by executing `$ python3 opcua.py -q "3.1.race" -c "ECC, Encrypt, no_reopen, SSec, cert, no_switch, no_leaks" --html`.

Results: `./reproduce_attacks.sh` takes less than 30 minutes to execute and stores results in several `output_<attack>` directories (where `attack` refers to the specific attack). The complete output can be browsed in `index.html` of that folder and attack traces are available in `tracei.pdf` (where `i` is an integer, typically `1` or `2`). Specifically, the race condition attack is found and reconstructed in less than 10 s, and depicted in the file `output/trace1.pdf` when produced with `opcua.py`.

(E2): [Confidentiality properties: $Conf_C$, $Conf_S$ & $Conf_{Pwd}$] [1 human-hour + 10 compute-hours]:

How to: The `results.md` file provides detailed instructions on how to launch ProVerif with the maximal configurations that back up our claim (C2).

Execution: Only the command for the first configuration for $Conf_C$ is shown here. The others are similar.

```
$ python3 opcua.py -q "Conf[C]" -c "RSA,
None|Sign|Encrypt, reopen, SNoAA|SSec,
anon|pwd|cert, switch, lt_leaks"
```

Results: All commands terminate with a ProVerif verification summary stating that the query is `true`.

(E3): [Weakened agreement properties: Agr_S^- and Agr_C^-] [1 human-hour + 24 compute-hours]:

How to: The `results.md` file provides detailed instructions on how to launch ProVerif with the maximal configurations that back up our claim (C3). In particular, it describes which invariants must first be proven and how.

Preparation: The property and its invariants to prove can be launched in parallel on a multi-core machine. Note however that some queries can be very demanding, for instance `3.1.B` in configuration ECC + Enc + Cert + Leak required 23 compute-hours and 30 GB of memory.

Execution: Only the command for proving the main property Agr_S^- without its invariants (called "3.1" in our model) is shown here: `$ python3 opcua.py -q "3.1" -c "RSA, Encrypt, no_reopen, SNoAA, cert, no_switch, lt_leaks"`. All other required invariants, according to `dependencies.txt`, must be proved with the same configuration.

Because this can be cumbersome, we additionally provide a helper script `reproduce_proofs.py` that will prove the property and all its invariants automatically. For example, the following command will fully prove Agr_S^- for the provided configuration: `$ python3 reproduce_proofs.py -q "Agr-[S->C]" -c "RSA, Encrypt, no_reopen, SNoAA, cert, no_switch, lt_leaks"`.

Results: All commands must terminate with a ProVerif verification summary stating that all queries are `true`. The first command proving `3.1` takes less than 3 seconds, the command proving Agr_S^- and its invariant concludes in less than 15 seconds. Some other properties and configurations from `results.md` are more computationally demanding, hence the overall time estimation.

(E4): [Launching lattice exploration campaigns] [15 human-minutes + 24 compute-hours]:

How to: The last section of `README.md` describes in details how to use `prove.sh` to launch complete lattice exploration campaigns.

Preparation: Lattice exploration campaigns will launch many different ProVerif runs and is thus best suited for a server with many cores and a lot of memory (see Appendix A.2.3).

Execution: A campaign on the property $Conf_C$ can be launched with `$ ./prove.sh "Conf[C]" "RSA, None|Sign|Encrypt, reopen, SNoAA|SSec, anon|pwd|cert, switch, lt_leaks"` (note the second argument that defines the maximal configuration the campaign aims for; here the maximal configuration without ECC). The script first asks for a starting configuration, press `Enter` to use the minimal one.

Results: The script displays: 1. "the minimal FALSE configurations" for which an attack was found, 2. "the minimal configurations" which could not be proved with the current resources budget (time and RAM), 3. "the maximal configurations" which were proven.

For the above command and after less than 20 core-minutes, the script will have ended the exploration and shows the proven maximal configuration (which was the maximal provided configuration).

Note that for other properties, some configurations may not be proven at all and the exploration will not terminate. We kill the script after some time, when we are able to extract the wanted information from the results.

A.5 Notes on Reusability

Our ProVerif models can be extended to explore new protocol evolutions and fixes. Our lattice exploration tool `prove.py` can be adapted to other protocols, see for example our generic lattice class `Trie` and `mark_trie` function.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Towards Internet-Based State Learning of TLS State Machines

Marcel Maehren[1], Nurullah Erinola[1], Robert Merget[2], Jörg Schwenk[1], and Juraj Somorovsky[3]

[1]Ruhr University Bochum
[2]Technology Innovation Institute
[3]Paderborn University

A Artifact Appendix

A.1 Abstract

In our main paper, we apply state machine learning to TLS deployments on the Internet and introduce techniques to address challenges arising from the uncontrolled environments. We further propose a novel methodology for an automated analysis of learned TLS state machines that allows us to identify vulnerabilities and non-compliant message flows. In a large-scale study, we collected 1304 state machines of real-world TLS hosts, uncovering several uncritical deviations, but also domains vulnerable to padding oracle attacks, and domains violating the session transcript integrity, potentially enabling MitM attacks.

This artifact appendix aims to show the functionality of the state learner we developed to conduct our study. The experiments involve the automated extraction of state machines from two versions of OpenSSL via Docker. First, we consider OpenSSL 3.4.0 a modern version exhibiting no state machine issues. Subsequently, we extract a state machine for OpenSSL 1.0.1j exhibiting an issue similar to the NetScaler transcript integrity vulnerability we discuss in our paper. For both versions, the respective experiment runs our automated analysis to test for issues. Finally, in a third experiment, we consider the analysis of anonymized state machines contained in our dataset to illustrate selected issues we observed in our evaluations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We are not aware of any security, privacy, or ethical concerns arising from running the experiments described below.

A.2.2 How to access

We provide our dataset via Zenodo.[1]

[1] https://doi.org/10.5281/zenodo.15520932

A.2.3 Hardware dependencies

For running the experiments, please ensure that the Java Virtual Machine can allocate at least 8GB of RAM.

A.2.4 Software dependencies

The experiments expect the ability to run bash scripts, and basic Linux CLI tools (wget/curl, awk, git, unzip, and grep). As part of the setup, Docker must be installed.

A.2.5 Benchmarks

Running the tool for experiment `E3` requires about 5 hours of computational time on a standard consumer grade desktop PC. Free disk space of 5 GB is required to build Docker images and for the dataset from our Zenodo repository.

A.3 Set-up

A.3.1 Installation

Please follow the instructions at https://docs.docker.com/get-started/get-docker/ to install Docker on your system. Please configure the permissions so that the user executing the scripts provided as part of this artifact is allowed to `docker build` and `docker run`. Subsequently, download and unzip the `artifact_experiments.zip` file from the Zenodo repository:

`https://doi.org/10.5281/zenodo.15520932`

Run the `setup.sh` script from the root directory of the artifact experiment directory. This script performs the following steps:

1. Downloads our source code from Zenodo.
2. Builds three Docker images: First, the script builds two images of OpenSSL 3.4.0 and OpenSSL 1.0.1j combined with our state learner tool. As part of this process, the source code fetched from Zenodo will be built using

Maven. Subsequently, the script builds a Docker image for our CLI tool (State Machine Analysis Tool).

3. Downloads our dataset from Zenodo.
4. Extracts the dataset to `experiment_dataset/`.
5. Creates output directories for experiments.
6. Executes a basic test to verify the setup (see Section A.3.2).
7. Asks if computational steps from E1 and E2 expected to take 5 hours and 10 minutes should be run immediately.

Note that you can skip the last step and perform the computational steps as part of the experiments later on.

A.3.2 Basic Test

The basic test is automatically executed as part of `setup.sh`. It runs the OpenSSL 3.4.0 container with minimal parameters to quickly verify functionality. The test:

1. Runs our tool to extract a basic state machine using Docker. As part of this process, the Docker container will open tmux with two panes showing OpenSSL on the left and our tool on the right. This test is expect to finish within 5 minutes. Results will be written to: `experiment_outputs/Basic_Test/alphabet-1/`
2. Verifies the creation of three output files:
   - `OpenSSL3.4.0.xml` - The state machine in XML format
   - `OpenSSL3.4.0_short.pdf` - A visualization of the state machine
   - `OpenSSL3.4.0_short.dot` - Analysis details in DOT format
3. Checks that the DOT file indicates that the analyses runs correctly

Expected output includes messages confirming successful Docker builds, dataset extraction, and verification that all expected files were created with correct content.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We claim to provide a tool to conduct state learning for TLS implementations. The tool iterates over different alphabets to yield increasingly detailed models of the state machine of an implementation (cf. Section 5 of our main paper). Throughout the execution, we employ a cache which achieves a high coverage rate (>=97% for the individual alphabets).

(C2): We claim to provide an automated analysis for the detection of state machine issues (cf. Section 5.6 of our main paper).

(C3): We claim to provide a data set documenting state machine issues. In particular, we claim to provide state machines showing hosts ignoring messages, invalid message paths leading to a completed TLS handshake, and padding-dependent behavior possibly enabling a padding oracle attack. These examples correspond to findings described in Sections 7.1.1 to 7.1.3 of our main paper.

A.4.2 Experiments

(E1): *Full execution for OpenSSL 3.4.0 [10 human-minutes + 5 compute-hours]: this experiment demonstrates state learning and analysis for a modern, correctly-behaving TLS implementation.*
Preparation: Ensure Docker images are built via `setup.sh`. If the 5-hour computational step was not run during setup, `E1.sh` will execute it.
Execution: Run `./experiments/E1.sh` from the repository directory. The script will:
- Check if the state machine already exists from setup
- If not, run the 5-hour learning process
- Execute the automated analysis
- Demonstrate tracing through the state machine with duplicate ClientHellos and unexpected certificates

Results: The automated analysis should report no issues beyond Internal Error alerts. The script demonstrates:
- Correct rejection of duplicate ClientHello with Unexpected Message alert
- Proper rejection of unsolicited client certificates
- A simplified visualization of the obtained state machine should further be available at `experiment_outputs/E1/alphabet-13/OpenSSL3.4.0_short.pdf`
- A simplified visualization for a smaller alphabet, resulting in a less extensive state machine, should further be available at `experiment_outputs/E1/alphabet-1/OpenSSL3.4.0_short.pdf`
- A log file providing stats for the extraction of the individual alphabets should be available at `experiment_outputs/E1/app.log`. For the individual alphabets, a high cache efficiency of >97% should be indicated.

(E2): *Illustrating issue detection based on OpenSSL 1.0.1j [10 human-minutes + 10 compute-minutes]: this experiment demonstrates the automated detection of state machine vulnerabilities in an older OpenSSL version. The issue is similar to the NetScaler real-world finding presented in the paper. Note that the execution is scoped to one alphabet to reduce the execution time.*

Preparation: Ensure Docker images are built via `setup.sh`. If the 10-minute computational step was not run during setup, `E2.sh` will execute it.
Execution: Run `./experiments/E2.sh` from the repository directory. The script will:

- Check if the state machine already exists from setup
- If not, run the 10-minute learning process (limited to the first alphabet as this suffices to illustrate the issue)
- Execute the automated analysis
- Demonstrate the issue by tracing duplicate ClientHellos
- Show complete invalid handshake paths involving duplicate ClientHellos

Results: The analysis identifies multiple issues:

- Invalid message paths with duplicate ClientHellos that complete the handshake
- Illegal inputs (like CCS after handshake) that do not trigger an error
- The script demonstrates how duplicate ClientHellos incorrectly receive ServerHello responses twice
- A simplified visualization of the obtained state machine should further be available at `experiment_outputs/E2/alphabet-1/OpenSSL1.0.1j_short.pdf`

(E3): *Inspecting key findings from our dataset [15 human-minutes]: this experiment analyzes representative state machines from our dataset to demonstrate selected issues we observed.*
Preparation: Ensure Docker images are built and dataset is extracted via `setup.sh`.
Execution: Run `./experiments/E3.sh` from the repository directory. The script analyzes three state machines:

- `completed-59.xml` - A NetScaler state machine which accepts duplicated ClientHello messages
- `completed-331.xml` - A state machine accepting unsolicited certificates
- `completed-1280.xml` - A state machine exhibiting deviating behavior for multiple padding oracle test vectors

Results: The script demonstrates:

- NetScaler accepting duplicate ClientHellos (similar to E2)
- Improper certificate handling allowing unrequested certificates to be sent in the handshake
- Padding oracle vulnerabilities with behavioral differences for different padding types
- Each analysis includes traces showing the specific message paths leading to issues

A.5 Notes on Reusability

Scope In both E1 and E2, we configure the Docker container to run our state learner tool to perform 20,000 random word queries per state when conducting the equivalence tests. In our study, we used 42,000 random queries. To conduct the experiment with the same extent of equivalence tests, the `-queries` parameter of the respective shellscript can be adjusted. For E2, we further limit the execution to the first alphabet. Deleting the `-alphabetLimit` parameter from the shellscript will result in a full execution. Note that extracting the full state machine of OpenSSL 1.0.1j takes significantly longer than extracting the state machine of OpenSSL 3.4.0.

Inspecting the Dataset To inspect more of the state machines we collected, please refer to any of the Docker run commands for the CLI tool given in `experiments/E3.sh` and adapt the file path (`-f`) to point to another state machine XML file.

Tools For applying our state learner to other targets, we recommend to use the flags from E1 or E2 as guidelines as these parameters reflect how we used the tool for our study. Additionally, both the state learner and the state machine analysis tool provide a brief help functionality to guide users through their features.

- **State Learner**: Access help by running the tool with the `-h` or `-help` flag. This displays all available command-line options, including configuration parameters for alphabet selection, learning algorithms, and output formats.
- **State Machine Analysis Tool**: Once in the interactive shell, type `help` to see all available commands. For detailed information about a specific command, use `help <command>`. Additionally, launching the tool with `-help` provides command-line usage options.

Building Without Docker Building the project outside of the Docker image should only require Maven and a Java Development Kit. Please note that Java 11 is required to run the resulting Jars as some dependencies used in our project are not compatible with newer Java versions.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Haunted by Legacy: Discovering and Exploiting Vulnerable Tunnelling Hosts

Angelos Beitis
DistriNet, KU Leuven
angelos.beitis@kuleuven.be

Mathy Vanhoef
DistriNet, KU Leuven
Mathy.Vanhoef@kuleuven.be

A Artifact Appendix

A.1 Abstract

For the artifact evaluation, we provide the ZMap scanning modules, which can be used to scan the Internet to identify vulnerable tunnelling hosts. We additionally provide the scripts to test whether individual hosts are potentially vulnerable to any of the identified vulnerabilities. Lastly, we give the attack scripts for the Tunnelling-Temporal Lensing (TuTL) attack. Each of the provided tools is accompanied by a README, which further explains how the tool works and how it can be used.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The `zmapv6-main.zip` is a software used for scanning the entire IPv4 address space, without requiring the hosting of an opt-out page or the maintenance of a blocklist. We do not recommend scanning the entire IPv4 address space for the functional evaluation; instead, you can use it to scan the individual addresses provided by us. These vulnerable hosts are under our control.

A.2.2 How to access

https://doi.org/10.5281/zenodo.15706733

A.2.3 Hardware dependencies

You will need access to a VM instance (e.g., AWS). This instance should have an IPv4 and IPv6 address which are globally routable, as well as disabled firewall rules (E.g., Your machine must be capable of sending and receiving TTL-expired packets).

A.2.4 Software dependencies

You need to have access to an Ubuntu machine which has IPv6 and IPv4 Internet access. We recommend AWS instances. Furthermore, ensure that there are no firewall restrictions potentially blocking traffic on your instance. Any additional dependencies are covered in the installation section.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

First, create a vulnerable IP6IP6 tunnelling interface:

```
chmod +x ip6ip6.sh
sudo ./ip6ip6.sh <IPV6_ADDR> \
<IPV6_ADDR_RANDOM>
```

To install ZMap, export the ZIP file and do:

```
sudo apt-get install build-essential cmake libgmp3-
dev gengetopt libpcap-dev flex byacc libjson-c-dev
pkg-config libunistring-dev
cmake .
make -j4
sudo make install
```

For `temp_lens.zip`, export the ZIP file and do:

```
sudo apt install tcpreplay
pip install netifaces
pip install scapy
```

For `tunneltester.zip`:

```
python3 -m venv venv
source venv/bin/activate
pip install wheel scapy==2.6.1
sudo su
source venv/bin/activate
```

A.3.2 Basic Test

Tunneltester: For tunneltester

```
sudo python3 tunnel_tester.py \
<INTERFACE> -t6 \
2600:1f18:2869:9b00:5b7b:92c9:d861:d151 \
-P6 <IPv6>
```

ZMap: For ZMap:

```
sudo zmap 34.171.178.123 -M ipip \
- -external-ipv4-address <IPv4_ADDRESS>
```

A.4 Evaluation workflow

A.4.1 Major Claims

Our artifacts provide the means to identify vulnerable hosts, either individual machines or multiple machines, through Internet-wide scans.

(C1): Our artifacts can successfully identify vulnerable tunnelling hosts.

(C2): Our attack script can successfully be used to conduct a tunnelled-temporal lensing attack as described in the paper. For the functional badge, the tunneled-temporal lensing script needs to be able to collect latencies and send packets that will eventually reach a host.

A.4.2 Experiments

Instructions on how to run each component.

(E1): *[Tunneltester] [5 human-minutes + 2 compute-minutes]:*
How to: The tunneltester script will detect two of our own hosts for the listed vulnerabilities in the paper. Replace <INTERFACE> with your machine's main interface, e.g., eth0, <PRIVATE_IPV4> with the private IPv4 address of your machine in case it is behind a NAT (can be left empty if this is not the case), and <PUBLIC_IPV4> and <IPv6> with your VM's global IPv4 and IPv6 addresses, respectively.
Execution: Run the following two commands:

```
sudo python3 tunnel_tester.py \
<INTERFACE> -t 3.90.110.118 -t6 \
2600:1f18:2869:9b00:5b7b:92c9:d861:d151 \
-s 172.31.29.15 -s6 \
2600:1f18:2869:9b00:5b7b:92c9:d861:d151 \
-p <PRIVATE_IPv4> -P <PUBLIC_IPv4> \
-P6 <IPv6>
```

```
sudo python3 tunnel_tester.py <INTERFACE> \
-t 34.171.178.123 \
-s 34.171.178.123 \
-p <PRIVATE_IPv4> -P <PUBLIC_IPv4>
```

Results: The script will output whether or not the host is vulnerable to a specific protocol. Specifically, you are looking to see the `VULNERABLE` text. The 3.90.110.118 host should be vulnerable to IPv6-based protocols (such as IP6IP6, GRE6 etc). AWS cannot send TTL-expired packets, meaning TTL-expired scans will not detect vulnerable hosts. The 34.171.178.123 host is vulnerable to IPv4-based protocols, but Google Cloud blocks the host from replying to ICMP Echo requests.

(E2): *[ZMap Scanning modules] [5 human-minutes + 2 compute-minutes]:*
How to: The ZMap Scanning module is used to identify vulnerable tunnelling hosts on the Internet. It can also be used to scan for an individual host. We provide a script named `zmap.sh`, which will go over some of the scanning modules to scan specific hosts under our control.
Execution: Run the following command:

```
chmod +x zmap.sh
sudo ./zmap.sh <YOUR_IPv4_ADDRESS>
<YOUR_IPv6_ADDRESS>
```

Results: The script goes over the scanning modules. You should see for each one the following:
`1 done (137 p/s avg); recv: 1 1 p/s`, indicating that the host scanned has been identified in the scan. For the final 4in6 scan, the scanning module will not detect the host since the inner source IP will be translated and the host does not spoof. To circumvent this, the script will catch the traffic in tcpdump. You should see the following:
`IP 3.90.110.118 > <YOUR_IPv4>: ICMP echo reply...`

(E3): *[Tunneled-Temporal Lensing Attack] [15 human-minutes + 10 compute-minutes]:*
How to: The tunnelled-temporal lensing attack can be used to concentrate traffic towards the attacker in time. For convenience, this script assumes the target is also a vulnerable tunnelling host. You will need to first create a vulnerable IP6IP6 interface, by running the script `ip6ip6.sh` as explained in Section A.3.
Execution: First create a vulnerable tunnelling interface by running:

```
sudo ./ip6ip6.sh <IPV6_ADDR> \
<IPV6_ADDR_RANDOM>
```

Where <IPV6_ADDR_RANDOM> is the IPv6 address for the newly created interface.
Prepare two new screens to capture the traffic:

```
screen -S outgoing
Press Ctrl+AD
screen -S incoming
Press Ctrl+AD
screen -S attack
Press Ctrl+AD
```

then create a file named `spoofers.txt` and add the two addresses listed below inside:

```
chmod +x create_spoofers.sh
sudo ./create_spoofers.sh \
2600:1f18:2869:9b00:bec4:73f4:9a4e:18ce \
2600:1f18:2869:9b00:21b7:96f:47b9:eed7
```

Run the following script. We set the sample size to 10. In the paper, the sample was 1000, but for convenience, we will use a short sample size of latencies in the artifact functionality assessment:

```
screen -r attack
sudo python3 -m src.main -i eth0 -vic
<IPV6_ADDRESS> -type ipip6 -alg bf -att 6
-sample 10 -spoo_f ./spoofers.txt
```

You should start seeing `Message Received!`. After latency collection is complete, you will be asked to start the attack. `Ready to attack? yes/no`: Do not press anything yet. Then do:

```
Press Ctrl+AD
screen -r outgoing
sudo ./outgoing.sh <IPv6 Address>
Press Ctrl+AD
screen -r incoming
sudo ./incoming.sh <IPv6 Address>
Press Ctrl+AD
screen -r attack
```

Now you will be back where you are asked to `Ready to attack? yes/no:`. Type yes and press enter. You can then press no to the following prompt.
The two files `outgoing.pcap` and `incoming.pcap` will be created after terminating the two scripts. These files have the incoming and outgoing traffic of the attack, respectively. This traffic can then be used for analysis in `wireshark`. Specifically, when opened in `wireshark`, you can go to `Statistics > I/O Graphs` and set the interval to 20ms.

A.5 Notes on Reusability

The addresses provided here are vulnerable and under our control and will be shut down after the artifact evaluation phase to prevent abuse.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: GeCos Replacing Experts: Generalizable and Comprehensible Industrial Intrusion Detection

Konrad Wolsing[*,‡] Eric Wagner[*,‡] Luisa Lux[*,‡] Klaus Wehrle[‡] Martin Henze[‡,*]

[*]*Fraunhofer FKIE* [‡]*RWTH Aachen University*

A Artifact Appendix

A.1 Abstract

Protecting Industrial Control Systems (ICSs) against cyber-attacks is crucial to counter escalating threats to critical infrastructure. To this end, Industrial Intrusion Detection Systems (IIDSs) provide an easily retrofittable approach to uncover attacks quickly and before they can cause significant damage. Current research focuses either on maximizing automation, usually through heavy use of machine learning, or on expert systems that rely on detailed knowledge of the monitored systems. While the former hinders the interpretability of alarms, the latter is impractical in real deployments due to excessive manual work for each individual deployment.

To bridge the gap between maximizing automation and leveraging expert knowledge, we introduce GeCo, a novel IIDS based on automatically derived comprehensible models of benign system behavior. GeCo leverages state-space models mined from historical process data to minimize manual effort for operators while maintaining high detection performance and generalizability across diverse industrial domains. Our evaluation against state-of-the-art IIDSs and datasets demonstrates GeCo's superior performance while remaining comprehensible and performing on par with rule-based IIDS based on the manual effort of experts.

This artifact provides the implementation of GeCo and describes how to reproduce our main results regarding the detection performance of GeCo, as well as our comparison to related work from Section 6.1 of the publication.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact does not contain content that threatens security or privacy as we publish the code and the evaluation tools for a defensive intrusion detection system (GeCo). GeCo is evaluated purely on datasets of ICS process data and thus our evaluation process does not require malware. Due to privacy concerns, we do not make the results of our user study from Sec. 6.2.2 available.

A.2.2 How to access

Our artifact is available at https://zenodo.org/records/15120036. Besides the files serving as an online appendix for our publication, *code-and-results.zip* contains the files for this artifact. It includes the code of the proposed IIDS (GeCo) and the configuration files used for its evaluation. A permanently maintained version of GeCo is available at the IPAL repository https://github.com/fkie-cad/ipal_ids_framework/tree/master/ids/geco, yet it is not required for reproducing the results from our publication.

A.2.3 Hardware dependencies

GeCo does not require specific hardware architectures. Still, the hardware requirements (CPU cores and memory) depend on the dataset and impact the runtime performance. To use and evaluate GeCo with pre-trained models, a single CPU core with a minimum of about 2Gb RAM has proven sufficient. However, to retrain the models of GeCo, more memory must and more CPU cores should be spared.

The smaller datasets (BATADAL and TEP) can be trained in under one hour with 8 CPU cores and at least 2Gb of memory. The training of the SWaT, WADI, and HAI models is much more demanding (cf. Section 6.3 and Table 5 in our paper). Regarding memory, they require at least 16GB for SWaT and HAI and 64GB for WADI to load the datasets. The number of CPU cores impacts the training time. Note that depending on the chosen number of CPU cores, additional memory is required to enable parallel processing.

A.2.4 Software dependencies

GeCo and the evaluation tools are a collection of Python scripts that have been tested with Python 3.12. Other versions are expected to work as well. The implementation has been tested on Linux and macOS operating systems.

The implementation of GeCo is based on the open-source IPAL IDS framework[1] and uses its provided datasets and evaluation tools. Except for the datasets (cf. the following A.2.5), all necessary software is provided within the artifact.

[1] IPAL – https://github.com/fkie-cad/ipal

List of used software versions:

- https://github.com/fkie-cad/ipal_ids_framework v1.5.2
- https://github.com/fkie-cad/ipal_evaluate v1.2.9
- https://github.com/fkie-cad/ipal_datasets v1.3.7

A.2.5 Benchmarks

Note that we cannot publicly provide the dataset files (*datasets/*) for legal reasons. Still, the datasets can be obtained by the publishers and transcribed into the IPAL format following the instructions here (cf. *datasets/README.md*). These steps are necessary to reproduce any of our results:

1. Clone the https://github.com/fkie-cad/ipal_datasets repository.
2. Install the dependencies listed in the *requirements.txt* file.
3. Select a dataset, e.g., SWaT, for this example.
4. Obtain the raw dataset from the publishers. Links to the dataset publishers are listed in the *ipal_datasets/README.md* table. The precise version and files used in our paper are listed below.
5. Place the dataset files in the respective folder. For example, *ipal_datasets/SWaT/raw/[dataset files]*.
6. Execute the *transcribe.py/sh* script for the chosen dataset.
7. Copy the generated files and replace this artifact's placeholders *attack.json* and *ipal/** files.

List of used files and dataset versions:

- SWaT: Download all three XLSX files from the *SWaT.A1 & A2_Dec 2015/Physical* folder.
- WADI: Download all CSV files from the *WADI.A2_19 Nov 2019* folder.
- HAI: Download all **csv.gz* from the https://github.com/icsdataset/hai/tree/master/hai-21.03 repository.
- BATADAL: Download all three CSV files from https://www.batadal.net/data.html (*BATADAL_dataset04.csv*, *BATADAL_dataset03.csv*, and *BATADAL_test_dataset.csv*).
- TEP: Download all files from the https://github.com/mikeliturbe/pasad/tree/master/data repository.

A.2.6 Description of files and content

The artifact consists of various files and folders, some relevant for reproducing our results while others serve as additional material for our paper. In the following, we describe the structure of the artifact:

- *code/*: This folder contains the software necessary for conducting our evaluation, based on the IPAL IDS Framework (cf. A.2.4), and the implementation of GeCo (*code/ipal-ids-framework/ids/GeCo*).
- *config/*: For each dataset, the **.json* files define the hyperparameters for GeCo. Based on these configurations and the training data, GeCo learns a model stored within the respective **.model* files.
- *datasets/*: We evaluated GeCo on five datasets stored in this folder (cf. A.2.5 on how to prepare the datasets).
- *knowledge-based/*: This folder contains the artifact to reproduce our results of Sec. 6.1.3.
- *output/*: Once the datasets are labeled by GeCo, we store the IDSs output here.
- *related-work/*: This folder contains the alerts of related work (SIMPLE, TABOR, Invariant, Seq2SeqNN, PASAD), which we use as a comparison in Table 2. These results were reproduced with the help of the IPAL IDS framework. The **.state.gz* files contain the labeled datasets for each IDS, and the **.json* files the respective metrics.
- *results*: Based on the labeled dataset from *output/*, the **.json* files contain the calculated performance metrics for GeCo and the **.pdf* files depict the alerts of GeCo (black) compared to the dataset's attacks (red).
- *videos/*: This folder contains renderings of the dependency graph from Figure 7a for all datasets, which are complementary material for our publication.

A.3 Set-up

A.3.1 Preparation

Please prepare the dataset files as described in A.2.5. For each dataset, all the dummy files in the respective dataset folders (*datasets/SWaT,WADI,HAI,BATADAL/**) have to be replaced.

A.3.2 Installation

Execute the following commands to install the required software for GeCo:

```
python3 -m venv venv
source ./venv/bin/activate
pip3 install igraph==0.10.4
pip3 install code/ipal-ids-framework/
pip3 install code/ipal-evaluate/
```

More information can be found in the *README.md* file.

A.3.3 Basic Test

Execute the following command to apply GeCo to one of the selected datasets.

```
./run-ids.py {SWaT,WADI,HAI,BATADAL,TEP}
```

To test whether the installation was successful, select "n" in the *run-ids.py* script to skip the sometimes long training process. For testing the training, use one of the datasets for which training is fast, such as BATADAL or TEP.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): GeCo shows competitive detection performance, which is often better than related work. This is proven by the evaluation (E1) described in Section 6.1.1, whose results are reported in Table 2, Table 6, and Figure 12.

(C2): GeCo generalizes well to a new domain of the chemical Tennessee Eastman Process (TEP). This claim is supported by the evaluation (E2) described in Section 6.1.2, whose results are reported in Figure 6.

(C3): GeCo yields a detection performance that is on par with labor-intensively created knowledge-based IIDSs. This claim is assessed in evaluation (E3) described in Section 6.1.3 and Table 3.

(C4): GeCo's alerts are comprehensible as they closely correlate to the ICS's underlying physical process. This experiment was designed and conducted in Section 6.2.1 and the results are shown in Figure 7 and Figure 8.

A.4.2 Experiments

How to, preparation, and execution

To conduct the evaluation, execute the *run-ids.py [dataset]* script, providing one of the five datasets (SWaT, WADI, HAI, BATADAL, and TEP) as the argument one by one. If the goal is to reproduce our results, select "n" at the beginning of the *run-ids.sh* script. That way, GeCo uses the provided pre-trained models from the *config/* folder. These steps suffice to reproduce the results of our experiments (E1–E4). [5 human-minutes, 10 compute-minutes, 1GB disk space]

Optionally, the detection models of GeCo can also be retrained on the training data instead of using the pre-trained models. To this end, select "y" at the beginning of the *run-ids.sh* script. Since the training phase can be extensive depending on the dataset, it is advisable to leverage an appropriate amount of CPU cores for parallelization (cf. Section 6.3). The number of cores for training can be configured in the configuration files under *config/{SWaT,WADI,HAI,BATADAL,TEP}.json* with the *cpus* option (default is 7). We recommend training the BATADAL or TEP model first, as they are the least computationally demanding datasets [5 human-minutes, 1-2 compute-hours, 1GB disk space]. SWaT, HAI [5 human-minutes, 24 compute-hours] and WADI [5 human-minutes, 1+ compute-month] demand significantly more time for training even with 64 CPU cores and 236GB of RAM. Yet, this step is not mandatory for the success of the following experiments.

Results

(E1): *Detection Performance* [10 human-minutes]: GeCo's detection performance is measured by comparing its alerts to the datasets labels. The results can be found in the *results/* folder for each dataset, where the **.json* files list different detection performance metrics and the **.pdf* files visualize the alerts in relation to the attacks. To visualize and aggregate these results, execute the respective scripts (*table-2.py*, *fig-12.py*, and *table-6.py*) to obtain the results shown in our publication. These scripts either render an image or generate the LaTeX code for the respective figures and tables.

(E2): *Generalizability* [5 human-minutes]: We show that GeCo successfully transfers to a new industrial domain by applying GeCo to the TEP dataset. The detection performance stated in Section 6.1.2 (96.0 in F1 and 98.0 in eTaF1) can be validated with the *results/TEP.json* file. Additionally, the *fig-6.py* script renders GeCo's alerts compared to the PASAD IIDS.

(E3): *Expert Knowledge-based IIDS* [5 human-minutes]: We implemented an IIDS that leverages a collection of Invariants as listed in Table 8. Execute the following script located under *knowledge-based/run.sh* to apply this IIDS to the SWaT dataset. The results are stored in the *output.json* and *output.pdf* files. To reproduce Table 3 of our paper, execute the *table-3.py* script.

(E4): *Alerts Localization and Understandability* [5 human-minutes]: We analyze GeCo's detection model to infer which process values are responsible for the alerts by analyzing the dependency between the learned state-space models by means of a dependency graph in Figure 7a. The dependency graph can be recreated with the respective *fig-7a.py* script. Note that the graph is arranged differently with each execution of the script. Lastly, the *fig-8.py* script generates the results for Figure 8.

A.5 Notes on Reusability

If the goal is to examine GeCo's detection performance in more detail on the datasets evaluated so far, please refer to the files in the *output/* directory, as they contain the raw output of GeCo for each dataset. For other activities, such as applying GeCo to different datasets, we recommend working with the implementation provided by the IPAL IDS framework[1]. To this end, we direct the reader to the tutorial and documentation about IPAL[2]. For guidance on how to configure the hyperparameters of GeCo, please refer to Section 7.1 of our publication.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

[2] IPAL Tutorial – https://github.com/fkie-cad/IPAL/blob/main/tutorial/Tutorial.md



USENIX Security '25 Artifact Appendix: Sometimes Simpler is Better: A Comprehensive Analysis of State-of-the-Art Provenance-Based Intrusion Detection Systems

Tristan Bilot[123†], Baoxiang Jiang[4†], Zefeng Li[5], Nour El Madhoun[2],
Khaldoun Al Agha[1], Anis Zouaoui[3], Thomas Pasquier[5]
[1]Université Paris-Saclay, [2]LISITE, Isep,[3]Iriguard,
[4]Xi'an Jiaotong University, [5]University of British Columbia

A Artifact Appendix

A.1 Abstract

In this study [3], we developed a unified framework that consolidates eight state-of-the-art PIDSs [4–10, 12] into a single codebase. This framework supports extensive customization, allowing users to integrate components from different systems for complex analyses and ablation studies. Where possible, the original code was optimized with GPU-accelerated operations and refactored according to best coding practices to improve maintainability. Each system is configured through its own YAML file, specifying system-specific parameters. To reduce redundancy, the framework employs a pipeline system that automatically reuses previously computed components and leverages existing results when available. The framework is primarily implemented in PyTorch and PyTorch Geometric and comprises 82 Python files totaling 14,811 lines of code.

We intend this project as a living artifact and invite the community to submit new systems.

A.2 Description & Requirements

All training and evaluation were conducted on a server running Ubuntu 22.04, equipped with a 3.2GHz 16-core AMD EPYC 7343 CPU, 1024 GB of memory, and an NVIDIA GA100 GPU with 80GB of memory.

A.2.1 Security, privacy, and ethical concerns

To the best of our knowledge, this work does not raise any ethical issues. All experiments have been performed on publicly available datasets that have been acquired in an ethical manner and do not contain any sensitive information. No security mechanisms are disabled, nor are any destructive actions performed during the evaluation.

[†]Work partially completed while at the University of British Columbia.

A.2.2 How to access

Permanent link: The source code is available on Zenodo: https://zenodo.org/records/15603122.
Archived Git repository: The original code corresponding to the paper is available at https://github.com/ubc-provenance/PIDSMaker/tree/velox.
Maintained version of the framework: The updated and maintained version of the framework is available at https://github.com/ubc-provenance/PIDSMaker.
Documentation: A detailed documentation is available at https://ubc-provenance.github.io/PIDSMaker/.

A.2.3 Hardware dependencies

Access to a relatively high-end GPU is recommended to speed up experiment completion.

A.2.4 Software dependencies

The experiments were conducted in the following environment.
OS. 5.19.0-46-generic #47~22.04.1-Ubuntu
Docker. 28.1.1, build 4eba377
CUDA. 12.2
NVIDIA driver. 535.230.02

The following steps apply only if Docker is not already installed and working properly with CUDA.

1. Install Docker following the official instructions and configure it to run without sudo.
2. Install dependencies for CUDA support with Docker:

```
# Add the NVIDIA package repository
curl -fsSL https://nvidia.github.io/
    ↪ libnvidia-container/gpgkey | sudo
    ↪ gpg --dearmor -o /usr/share/keyrings
    ↪ /nvidia-container-toolkit-keyring.
    ↪ gpg
```

```
curl -s -L https://nvidia.github.io/
    ↪ libnvidia-container/stable/deb/
    ↪ nvidia-container-toolkit.list | \
  sed 's#deb␣https://#deb␣[signed-by=/usr/
      ↪ share/keyrings/nvidia-container-
      ↪ toolkit-keyring.gpg]␣https://#g' |
      ↪  \
  sudo tee /etc/apt/sources.list.d/nvidia-
      ↪ container-toolkit.list

# Update and install
sudo apt-get update
sudo apt-get install -y nvidia-container-
    ↪ toolkit

# Restart services
sudo systemctl restart docker
sudo nvidia-ctk runtime configure --runtime
    ↪ =docker
sudo systemctl restart docker
```

A.2.5 Benchmarks

We use the DARPA E3 [1], E5 [2], and OpTC [11] datasets.

A.3 Set Up

Note: If encountering issues during the installation or reproducibility of experiments, please refer to the guidelines in the archived Git repository and consider opening an issue.

A.3.1 Installation

1. Clone the Repo

```
git clone https://github.com/ubc-provenance/
    ↪ PIDSMaker.git -b velox velox
cd velox
```

2. Download Datasets

The DARPA Transparent Computing (TC) and Operationally Transparent Cyber (OpTC) datasets are large and require substantial computational resources to process. We provide pre-processed versions of these datasets, stored in a PostgreSQL database, with database dumps available for download.

Datasets Size. The sizes of each database dump are as follows: *raw* refers to the size of the dump after downloading and uncompressing the archive, while *loaded* indicates the size once loaded into the PostgreSQL table.

Downloadable Archives. We provide three files:

1. `optc_and_cadets_theia_clearscope_e3.tar`, containing all OpTC datasets, CADETS_E3, THEIA_E3, and CLEARSCOPE_E3.

Dataset	Raw (GB)	Loaded (GB)
CLEARSCOPE_E3	0.6	4.8
CADETS_E3	1.4	10.1
THEIA_E3	1.1	12
CLEARSCOPE_E5	6.2	49
CADETS_E5	36	276
THEIA_E5	5.8	36
OPTC_H051	1.7	7.7
OPTC_H_501	1.5	6.7
OPTC_H201	2.0	9.1

Table 1: Raw and loaded sizes of DARPA TC and OpTC dataset dumps.

2. `theia_clearscope_e5.tar`, containing `THEIA_E5` and `CLEARSCOPE_E5`.

3. `cadets_e5.dump` is the dump for CADETS_E5 (very large).

Steps.

1. Download the archive(s) into a new `data` folder from Google Drive. On the command line, use `curl` with an authorization token (see Stack Overflow for details):

   - Visit the OAuth 2.0 Playground at developers.google.com/oauthplayground.
   - In the `Select the Scope` box, paste `https://www.googleapis.com/auth/drive.readonly`.
   - Click `Authorize APIs`, then `Exchange authorization code for tokens`.
   - Copy the **Access token**.
   - Run the following in a terminal:

   Note. Each `curl` call may download only part of the file before pausing. When the loading bar stops, press `Ctrl+C` and rerun the command until the download completes.

```
mkdir data && cd data

# OpTC and E3 Datasets
curl -H "Authorization:␣Bearer␣
    ↪ ACCESS_TOKEN" -C - https://www.
    ↪ googleapis.com/drive/v3/files/11
    ↪ YVPAuWfeEqC_zV8KD0gNrnEPbHf2Y4M?alt=
    ↪ media -o
    ↪ optc_and_cadets_theia_clearscope_e3.
    ↪ tar

# THEIA and CLEARSCOPE E5 Datasets
```

```
curl -H "Authorization:␣Bearer␣
    ↪ ACCESS_TOKEN" -C - https://www.
    ↪ googleapis.com/drive/v3/files/1
    ↪ DfolzEa3PVz_6fGZUNEUm0sBP42LB7_1?alt
    ↪ =media -o theia_clearscope_e5.tar

# CADETS E5 Dataset
curl -H "Authorization:␣Bearer␣
    ↪ ACCESS_TOKEN" -C - https://www.
    ↪ googleapis.com/drive/v3/files/1
    ↪ Xiq7w0Ofz4jZG2PVFuNqi_i0fm28kRcT?alt
    ↪ =media -o cadets_e5.dump
```

2. Uncompress the archives (requires minimal additional space):

```
tar -xvf
    ↪ optc_and_cadets_theia_clearscope_e3.
    ↪ tar
tar -xvf theia_clearscope_e5.tar
```

3. Load Databases

We create two containers: one for the PostgreSQL database and another to execute the experiments.

1. Set your paths in `.env`:

```
cp .env.local .env
```

In `.env`, set `INPUT_DIR` to the `data` folder path. Optionally, set `ARTIFACTS_DIR` to the folder for generated files (multiple GBs). Then run:

```
source .env
```

2. Build and start the database container:

```
docker compose -p postgres -f compose-
    ↪ postgres.yml up -d --build
```

Note: Update environment variables using `source .env` after modifying `.env` before running `docker compose`.

3. In a terminal, access a shell in the container:

```
docker compose -p postgres exec postgres
    ↪ bash
```

4. If you have sufficient storage (135 GB), load all databases:

```
./scripts/load_dumps.sh
```

For limited storage, load databases individually:

```
pg_restore -U postgres -h localhost -p 5432
    ↪ -d DATASET /data/DATASET.dump
```

5. Once loaded, exit the container:

```
exit
```

4. Get into the PIDSMaker Container

The `pids` container is used for development and to run experiments.

1. For VSCode users, use the Dev Containers extension. Install the extension, then press `Ctrl+Shift+P` and select *Dev Containers: Open Folder in Container*.

2. Alternatively, load the container manually:

```
docker compose -f compose-pidsmaker.yml up
    ↪ -d --build
docker compose exec pids bash
```

The Python environment and framework are installed in this container. Now download the weights necessary to reproduce the Velox results.

```
pip install gdown
cd weights

# Neural net encoder weights
gdown https://drive.google.com/drive/folders/18
    ↪ vc5JuMwUY2TmrlONG3xyPdh9NfBXyAD --folder

# Word2vec weights
gdown https://drive.google.com/drive/folders/1w
    ↪ -QoUYsnAKVVMLnLfxQ2qS1W5--1kUaD --folder
```

A.3.2 Basic Test

Once you have a shell in the `pids` container, execute the following commands:

```
cd scripts
./run_local.sh velox THEIA_E3 --tuned
```

A.4 Evaluation workflow

A.4.1 Major Claims

The paper is experiment-heavy. It represents a high computational cost (a total of 453 days of computation on our hardware). Consequently, we do not expect you to reproduce all results presented in the paper. Instead we focus on the two following major claims:

(C1): A simpler ML architecture can achieve state-of-the-art performance on a number of common benchmark. This is demonstrated in Table 4, 5, and 6 via the ADP Best metric, and can be reproduced by the experiment E1.

(C2): We notice a high instability in the results. Training the system from scratch will lead to significantly different results across runs. This is discussed in §4, SC5. This is demonstrated in Table 4, 5, and 6 via the $\tilde{\sigma}_{\text{ADP}}$ metric, and can be reproduced by the experiment E2.

A.4.2 Experiments

Due to high computation cost, we focus on reproducing results from `velox`, `orthrus` and `nodlink` shown to have reasonable computational cost (see §5.1, Figure 13).

(E1): [5 human-minutes + few hours of compute + $\sim$ 300 GB disk]:

How to: We provide a single script (run_local.sh) that reproduces the final experimental results (Tables 4, 5, 6) presented in the paper. Based on claim C1, we only want to reproduce the best ADP metrics, which indicate the best-case detection capability of a system.

Preparation: Follow steps described in §A.3.

Execution: To reproduce best ADP scores, we provide the pre-computed weights, as training can take up to multiple days:

```
./run_local.sh velox DATASET --from_weights
    ↪ --tuned
```

- Replace DATASET with one of the following: `THEIA_E3` `CADETS_E3`, `CLEARSCOPE_E3`, `CLEARSCOPE_E5`, `CADETS_E5`, `THEIA_E5`, `optc_h201`, `optc_h501`, or `optc_h051`.

Results: Compare the `adp_score` metric with the associated value in Tables 4, 5, 6. Note that the ADP scores might be slightly different due to important instability, as demonstrated in experiment E2.

(E2): [5 human-minutes + several days of compute + $\sim$ 1TB disk]:

How to: Based on claim C2, each run spans for five iterations and the Mean, Min and Best metrics are printed, along with the standard deviation. For C2, we only focus on the standard deviation to measure the instability of systems. This is measured by the $\tilde{\sigma}_{\text{ADP}}$ metric.

Preparation: Follow steps described in §A.3.

Execution: The goal is to reach high $\tilde{\sigma}_{\text{ADP}}$ scores, as in Tables 4, 5, 6:

```
./run_local.sh {system} DATASET --
    ↪ experiment=run_n_times --tuned
```

- Replace `{system}` with one of the following: `velox`, `orthrus`, and `nodlink` (additionally `thretrace`, `kairos`, `rcaid`, and `flash` are also available).

Example with `velox`:

```
./run_local.sh velox THEIA_E3 --experiment=
    ↪ run_n_times --tuned
./run_local.sh velox CADETS_E3 --experiment
    ↪ =run_n_times --tuned
./run_local.sh velox CLEARSCOPE_E3 --
    ↪ experiment=run_n_times --tuned
```

Results: Compare the `adp_score_std_rel` metric with $\tilde{\sigma}_{\text{ADP}}$ in Tables 4, 5, 6. Here we show that all systems are subject to instability. This means the measured performance of a system differs significantly from one run to another. Consequently, **exact reproduction of the original results is very unlikely**. This is not a bug and is indeed the expected behavior (see §4, SC5).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] Transparent Computing Engagement 3 Data Release, Accessed 10th August 2025. https://github.com/darpa-i2o/Transparent-Computing/blob/master/README-E3.md.

[2] Transparent Computing Engagement 5 Data Release, Accessed 10th August 2025. https://github.com/darpa-i2o/Transparent-Computing.

[3] Tristan Bilot, Baoxiang Jiang, Zefeng Li, Nour El Madhoun, Khaldoun Al Agha, Anis Zouaoui, and Thomas Pasquier. Sometimes Simpler is Better: A Comprehensive Analysis of State-of-the-Art Provenance-Based Intrusion Detection Systems. In *Security Symposium (USENIX Sec'25)*. USENIX, 2025.

[4] Zijun Cheng, Qiujian Lv, Jinyuan Liang, Yan Wang, Degang Sun, Thomas Pasquier, and Xueyuan Han. Kairos: Practical Intrusion Detection and Investigation using Whole-system Provenance. In *Symposium on Security and Privacy (S&P'24)*. IEEE, 2023.

[5] Akul Goyal, Gang Wang, and Adam Bates. R-CAID: Embedding Root Cause Analysis within Provenance-based Intrusion Detection. In *Symposium on Security and Privacy (S&P'24)*. IEEE, 2024.

[6] Xueyuan Han, Xiao Yu, Thomas Pasquier, Ding Li, Junghwan John Rhee, James W. Mickens, Margo I. Seltzer, and Haifeng Chen. SIGL: Securing Software

Installations Through Deep Graph Learning. In *Security Symposium (USENIX Sec'21)*. USENIX, 2021.

[7] Zian Jia, Yun Xiong, Yuhong Nan, Yao Zhang, Jinjing Zhao, and Mi Wen. MAGIC: Detecting Advanced Persistent Threats via Masked Graph Representation Learning. In *Security Symposium (USENIX Sec'24)*. USENIX, 2024.

[8] Baoxiang Jiang, Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, Anis Zouaoui, Shahrear Iqbal, Xueyuan Han, and Thomas Pasquier. ORTHRUS: Achieving High Quality of Attribution in Provenance-based Intrusion Detection Systems. In *Security Symposium (USENIX Sec'25)*. USENIX, 2025.

[9] Shaofei Li, Feng Dong, Xusheng Xiao, Haoyu Wang, Fei Shao, Jiedong Chen, Yao Guo, Xiangqun Chen, and Ding Li. NODLINK: An Online System for Fine-Grained APT Attack Detection and Investigation. In *Network and Distributed System Security Symposium (NDSS'24)*. The Internet Society, 2024.

[10] Mati Ur Rehman, Hadi Ahmadi, and Wajih Ul Hassan. Flash: A Comprehensive Approach to Intrusion Detection via Provenance Graph Representation Learning. In *Symposium on Security and Privacy (S&P'24)*. IEEE, 2024.

[11] Mike van Opstal and William Arbaugh. Operationally Transparent Cyber (OpTC) Data Release, 2019. https://github.com/FiveDirections/OpTC-data.

[12] Su Wang, Zhiliang Wang, Tao Zhou, Hongbin Sun, Xia Yin, Dongqi Han, Han Zhang, Xingang Shi, and Jiahai Yang. Threatrace: Detecting and tracing host-based threats in node level through provenance graph learning. *IEEE Transactions on Information Forensics and Security*, 17:3972–3987, 2021.



USENIX Security '25 Artifact Appendix: CAMP in the Odyssey: Provably Robust Reinforcement Learning with Certified Radius Maximization

Derui Wang[♡,♠], Kristen Moore[♡,♠], Diksha Goel[♡,♠], Minjune Kim[♡,♠], Gang Li[♣], Yang Li[♣], Robin Doss[♣], Minhui Xue[♡,♠], Bo Li[◊], Seyit Camtepe[♡,♠], and Liming Zhu[♡]

[♡]*CSIRO's Data61, Australia*
[♠]*Cyber Security Cooperative Research Centre, Australia*
[♣]*Deakin University, Australia*
[◊]*University of Chicago, USA*

A Artifact Appendix

A.1 Abstract

This artifact appendix focuses on two main claims (please refer to "Contributions" in Section 1 and "Main takeaways" at the end of Section 5) in our paper:

- **`CAMP` and *policy imitation* enhance the certified robustness of DRL agents:** We compare `CAMP` with two other baseline training methods, namely Gaussian and NoisyNet. `CAMP` outperforms these baselines in the trade-off between certified agent utility (*i.e.,* "Certified Expected Return") and certified robustness (*i.e.,* "Certified Radius"). In particular, significant performance gains are observed in both classic control and autonomous driving environments.
- **Agents trained by `CAMP` and *policy imitation* are more robust against empirical attacks:** `CAMP` effectively mitigates adversarial perturbations in observations, maintaining a higher average return across diverse attack scenarios. The `CAMP` agents exhibit superior robustness compared to Gaussian agents, with particularly significant improvements observed in classic control and autonomous driving environments.

We have provided the source code and scripts in this artifact to reproduce the results that support these two claims.

A.2 Description & Requirements

The artifact enables the demonstration and reproduction of results for both `CAMP`, Gaussian, and NoisyNet agents in the Cartpole environment. Since the original code and scripts were developed and tested on our internal high-performance computer (HPC) cluster (CSIRO Virga), which restricts access from external users, we have done our best to ensure the results can be reproduced on non-HPC devices. We provided a list of requirements for running our code on a customer PC to ensure that evaluators can conduct the evaluation.

A.2.1 Security, privacy, and ethical concerns

To the best of our knowledge, there are no security, privacy, or ethical concerns associated with our artifact. Our code does not collect data or fingerprints from evaluators. Moreover, these concerns are not applicable if the evaluator uses their own hardware and software platform for evaluation.

A.2.2 How to access

The code is available from our GitHub repository (https://github.com/NeuralSec/camp-robust-rl). Evaluators can simply clone the repository to their local PC and set up a virtual environment as instructed. Since training the DQN can be highly unstable, the quality of agents trained with different GPUs or in different system environments may vary. To ensure reproducibility, we set up containers with NVIDIA RTX-4090 on `runpod.io` , allowing evaluators to log in anonymously and run the corresponding experiments. The Zenodo release of our code can be accessed from https://zenodo.org/records/14729675.

A.2.3 Hardware dependencies

We recommend that evaluators use a PC equipped with a CUDA-capable GPU. While our code and scripts were developed and tested on an internal HPC cluster with NVIDIA H100 GPUs, the code is not memory-intensive. Consumer-grade GPUs supporting CUDA should handle execution without issues. For reference, training with `CAMP` and policy imitation on the Cartpole environment typically consumes less than 1GB VRAM.

Table 1: Environment for Experiments

Software	Detail
Operating system	Linux (SUSE Linux Enterprise Server SLE15)
Python version	3.11.4
CUDA version	12.4
cuDNN version	9.1.0
Major libraries	Pytorch 2.5.1 Gymnasium 0.29.1 Highway-env 1.9.1 ale-py 0.8.1 autoattack 0.1 SciPy 1.11.0 Numpy 1.26.4 statsmodels 0.14.2 matplotlib 3.8.4 tensorboard 2.16.2 tqdm 4.66.4 mlconfig 0.1.8

A.2.4 Software dependencies

The code should be executable on various versions of Linux operating systems. A `requirements.txt` file is provided for quick setup of the environment. This file includes a comprehensive list of dependencies, rather than just the minimal required ones, to facilitate the recreation of the virtual environment used during our experiments. Table 1 lists the system environment and major libraries used during our code testing.

A.2.5 Benchmarks

We mainly produce results based on the Gymnasium [3] Cartpole environment in this artifact. Specifically, this artifact uses `"Cartpole-v0"` from Gymnasium. The Q-network of the agent is a multi-layer perception (MLP) whose architecture is described in Table 4 of our paper.

A.3 Set-up

- Clone the repository from GitHub to your local PC or the provided container.
- A `readme.md` file included in the repository describes the commands for environment setup.
- The experiment will create new folders within the current directory to store its results. Please ensure that the directory where you run the experiment has the necessary write permissions to allow the creation of these folders.
- We provide four shell scripts to help you reproduce the results.

A.3.1 Installation

The installation steps are described in the `readme.md` file. First, navigate to the root directory of the cloned repository and create a virtual environment using `venv` . Next, update `pip` to the latest version and install dependencies from `requirements.txt` . After installation, you can type `deactivate` to exit the virtual environment. To this end, the setup is finished and the code should be ready for the following evaluation. We have observed that when installing dependencies on some Ubuntu distributions (tested on WSL2 Ubuntu 24.04.1 LTS), running `pip install -r requirements.txt --no-cache-dir` may result in errors like segmentation fault due to conflicting dependency versions. If this occurs, rerunning the same command immediately after the error often resolves the issue.

A.3.2 Basic Test

In the root directory of the cloned repository, execute `bash ae_step1.sh` . If the Python version and GPU information are displayed without errors, the setup is successful. To monitor training with TensorBoard, open a new terminal, navigate to the root directory of the repository with the virtual environment activated, and run `tensorboard --logdir="exp"` . This will allow you to visualize the training progress.

A.4 Evaluation workflow

This artifact includes four shell scripts located in the root directory, which generate results for two major experiments designed to verify our two claims. In this evaluation, we will focus on reproducing results in the Cartpole environment under the setting of $\sigma = 0.2$.

- `ae_step1.sh` trains agents using `CAMP`, Gaussian, and NoisyNet on `"Cartpole-v0"` with single-frame observations (*i.e.,* "Cartpole-1" in the paper) and $\sigma = 0.2$.
- `ae_step2.sh` tests the three agents trained by `ae_step1.sh` and certifies their robustness. The results are plotted as a `.pdf` figure and saved in the directory `./exp_cert/comparisons/` .
- `ae_step3.sh` attacks the `CAMP` and Gaussian agents trained by `ae_step1.sh` using projected gradient (PGD) attack [2] and plots the result in a `.pdf` figure in the `./attacks/` folder.
- `ae_step4.sh` attacks the `CAMP` and Gaussian agents trained by `ae_step1.sh` using Auto PGD (APGD) attack [1] and plots the result in a `.pdf` figure in the `./attacks/` folder.

The results from `ae_step1.sh` and `ae_step2.sh` support our first claim, while those from `ae_step3.sh` and `ae_step4.sh` support our second claim.

A.4.1 Major Claims

(C1): *`CAMP` and policy imitation enhance the certified robustness of DRL agents. This is demonstrated by the experimental results in Figures 2 and 8 of the paper.*

(C2): *`CAMP` and policy imitation effectively improves the robustness of agents against empirical attacks perturbing*

observations. This is demonstrated by the experimental results in Figures 3 and 4 of the paper.

A.4.2 Experiments

(E1): *[Train agents with different methods on Cartpole-1 and certify the robustness of trained agents] [1 human-minute + 2.5 compute-hour]: Train agents with different methods and certify their robustness to support* ***(C1)****.*
How to:
Preparation: *Complete installation described in Section A.3 first.*
Execution: *1) In the root directory of the cloned repository, run* `bash ae_step1.sh` *. The script trains* `CAMP`*, Gaussian, and NoisyNet agents in the Cartpole-1 environment with the noise scale* $\sigma = 0.2$*. The trained agents are saved under the* `./exp/` *folder. 2) When finished, run* `bash ae_step2.sh` *. This script loads the trained agents and tests each agent in Cartpole-1 for* $10,000$ *independent episodes. The test rewards will be saved under* `./eval_exp/` *. Then the test rewards will be loaded and used for robustness certification. The plotted certification results will be saved in the directory* `./exp_cert/comparisons/` *.*
Results: *The plotted results should closely resemble those in Figure 2 (sub-figure "Cartpole-1" when* $\sigma = 0.2$*) of the paper. The certified curve for* `CAMP` *should be higher than those of the Gaussian and NoisyNet baselines.*

(E2): *[Apply PGD and APGD attacks to evaluate empirical robustness.] [1 human-minute + 2 compute-hour]: Compare the robustness of* `CAMP` *agent and Gaussian agent to support* ***(C2)****.*
How to:
Preparation: *Complete the installation described in Section A.3 and* ***(E1)*** *to ensure that trained agents are in place.*
Execution: *1) In the root directory of the cloned repository, run* `bash ae_step3.sh` *. This script attacks previously trained* `CAMP` *and Gaussian agents using PGD across different perturbation budgets. The attack results will be saved under* `./exp/` *. The comparison between* `CAMP` *and Gaussian agents will be plotted in* `attacks/cartpole_simple_pgd_threshold=0.pdf` *. 2) Run* `bash ae_step4.sh` *. This script attacks previously trained* `CAMP` *and Gaussian agents using APGD across different perturbation budgets. The attack results will again be saved under* `./exp/` *. The comparison between* `CAMP` *and Gaussian agents will be plotted in* `cartpole_simple_apgd_threshold=0.pdf` *.*
Results: *To reduce time costs for evaluators, we have decreased the number of evaluations from 1,000 to 100 when computing the average return in this artifact evaluation. The expected results should reflect the trends in Figures 3 and 4 (sub-figure "Cartpole-1" when* $\sigma = 0.2$*). The average return of the* `CAMP` *agent should be significantly higher than that of the Gaussian agent.*

A.5 Notes on Reusability

We have included four scripts in our code repository to enable a fuss-free reproduction of the training-testing-certification pipeline and empirical robustness evaluation tasks in "Cartpole-1" under a specific hyper-parameter setting ($\sigma = 0.2$). This configuration represents one of the major improvement cases (*i.e.,* "Cartpole" and "Highway"), and other configurations can be evaluated by simply setting the `env_id` and `env_sigma` arguments in the scripts. Beyond these scripts, detailed instructions for running experiments in different environments with various hyper-parameter settings are available in the `readme.md` file. This also enables others to adapt `CAMP` and policy imitation to agents and environments beyond those tested in our paper. For comprehensive information and access to the source code, please visit our GitHub repository: https://github.com/NeuralSec/camp-robust-rl.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2025/`.

References

[1] Francesco Croce and Matthias Hein. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *ICML*, 2020.

[2] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[3] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulao, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.



USENIX Security '25 Artifact Appendix: Robustifying ML-powered Network Classifiers with PANTS

Minhao Jin
Princeton University

Maria Apostolaki
Princeton University

A Artifact Appendix

A.1 Abstract

PANTS is a practical framework for assessing and enhancing the robustness of ML-based network traffic classification (MNC) using adversarial, realizable, and semantics-preserving inputs. This artifact includes all the necessary code and scripts for PANTS, along with supplementary resources such as datasets and well-trained models including both vanilla and robustified ones. Additionally, it provides a comprehensive roadmap for effectively utilizing and evaluating PANTS.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The execution of artifacts will not raise any risk on security, privacy, or ethical concerns. All the datasets required to evaluate artifacts are public.

A.2.2 How to access

To better evaluate PANTS for its functionality and reproducibility, we provide an instance in CloudLab that has already installed all the required dependencies, data, code, and scripts such that the user can test PANTS smoothly. For any reviewer who would like to use our provided instance, please send your public key via comments so that we can add you to the access list for the provided machine. After we add the public key, reviewers can easily access via

```
$ ssh minhaoj@nfs.usenix-artifact.netsyn.emulab.net
```

After logging into the instance, the PANTS codebase is in the directory `/nfs/PANTS`. Please note that the CloudLab instance will be terminated after the artifact evaluation period. However, the artifact codebase as well as the corresponding data and models will be open to the public permanently.

If reviewers prefer to evaluate locally, the latest version of the source code is provided in the https://github.com/jinminhao/PANTS. For artifact evaluation purposes, reviewers can refer to the latest commit. Reviewers need to download some secondary artifacts, such as datasets and trained models, from an external Google Drive link. Details of downloading these secondary artifacts will be introduced in §A.3. In the provided CloudLab instance, all these files are already downloaded.

We release the finalized version in both https://github.com/jinminhao/PANTS and https://zenodo.org/records/14756845.

A.2.3 Hardware dependencies

If the reviewer prefers to test locally, we recommend evaluating PANTS using the machine that has similar or identical performance as c220g5 or d430 provided by CloudLab. For example, the c220g5 machine has two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz with 192GB ECC DDR4-2666 Memory.

Please refer to the CloudLab hardware page (https://docs.cloudlab.us/hardware.html) for more details on the hardware specification.

A.2.4 Software dependencies

To execute the artifacts, the testbed must be running on a Linux operating system. Additionally, Conda must be installed on the testbed. All required software dependencies are listed in the environment specification files (requirements-app-vpn.txt and requirements-vca.txt), which is provided within the codebase along with detailed instructions on how to execute these scripts. We provide a detailed guide on installation in §A.3.1.

A.2.5 Benchmarks

We evaluate PANTS with three different applications, which would require three datasets. All the datasets and the corresponding well-trained models (including vanilla and robustified ones) have been provided via a shared google drive folder. Please refer to §A.3.1 for details on downloading them.

A.3 Set-up

A.3.1 Installation

Please note that installation is only required if reviewers are evaluating PANTS locally. For any evaluation using the pro-

vided machine, the environment has already been configured and it is not necessary to re-install.

Installation of PANTS basically includes three steps:

- Step 1: Make sure the conda is correctly installed. Please refer to the conda page for more details.
- Step 2: Download the asset from this Google Drive link and decompress the `asset.tar.gz`. After decompression, you should see an 'asset' directory which includes data and models.
- Step 3: Run the following code to install all the required dependencies. Basically, running the artifacts require two different virtual environments.

```
$ cd PANTS/
$ conda create -n py39-app-vpn python=3.9
$ conda activate py39-app-vpn
$ pip3 install -r requirements-app-vpn.txt
$ conda create -n py39-vca python=3.9
$ conda activate py39-vca
$ pip3 install -r requirements-vca.txt
```

A.3.2 Basic Test

After all the dependencies are installed, reviewers can easily test PANTS by running

```
$ cd PANTS/scripts/
$ bash test-env.sh
```

To briefly introduce, this script uses PANTS to attack two models under two different applications (APP and QOE). If the environment is successfully configured, it should return with outputs like the following:

```
$ bash test-env.sh

...
Summary: ASR: 1.0, speed: 2.0500868564283423
...
Summary: ASR: 1.0, speed: 0.5714318749236006
```

Please note that the script will automatically activate the required environment based on the target application, i.e., when PANTS is running against APP(QOE), it will switch to the environment py39-app-vpn(py39-vca) automatically.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PANTS is 70% more likely to find adversarial samples compared to Amoeba in median and 2x more likely compared to BAP. This is reflected in Fig. 6 in Finding 1. It describes the ASR when using PANTS and baselines to find adversarial samples, and it shows that PANTS can find adversarial samples with a high success rate.

(C2): Iterative augmentation with adversarial, realizable, and semantics-preserving samples improves the robustness of an MNC without hurting its accuracy. It corresponds to Fig. 7 and Fig. 8 in Finding 2. Fig. 7 describes the accuracy and ASR for models robustified by different approaches. It shows that PANTS can robustify models without sacrificing model accuracy. Fig. 8 plots the improvement of ASR during iterative robustification using PANTS.

(C3): PANTS improves the robustness of the tested MNCs 142% more than Amoeba and 40X more than NetShare. This claim is in the Finding 3. The corresponding figures are Fig. 9 and Fig. 10. Fig. 9 shows the accuracy and ASR when leveraging NetShare for robustfication. Fig. 10 describes the ASR for vanilla, PANTS-robustified and Amoeba-robustified models.

(C4): PANTS can improve the robustness of MNCs even against threat models outside those used during adversarial training. Fig. 11 in Finding 4 shows the ASR of two APP implementations for various threat models.

(C5): PANTS is practical and efficient to use, generating ∼1.7 samples/sec, which is 8X faster than Amoeba. Fig. 12 in Finding 5 plots the generation speed for PANTS and Amoeba.

(C6): Transferability provides an opportunity to robustify an MNC without white-box access. This is mainly discussed in Finding 6. Fig. 13 presents the result of transferability.

A.4.2 Experiments

Strictly running all the experiments would require weeks on a 5-machine cluster. Therefore, we only report PANTS' result on a **subset** of data for each experiment, which still needs ∼46 compute-hours when running on our provided instance. As baselines need to train on a full set of data, which is very time-consuming, evaluation of them is not included in this artifact. Reviewers can refer to the results in the paper for their performance.

(E-all): [1 human-minute + 46 computer-hours + 20GB disk]:

How to: Reviewers can run all the experiments using a single script. The script automatically runs all the provided testing scripts from (E1) to (E5) to get the results and generate the corresponding figures.

In short, it is easy to run (E-all) by

```
$ cd PANTS/scripts
$ bash test-all.sh
```

Results: All the results are presented as figures in the directory `PANTS/figures`. For each generated figure, please refer to the following (E_i) for a detailed explanation.

(E1): [1 human-minute + 25 compute-hour + 20GB disk]: (E1) generally includes 2 parts. First, it evaluates

PANTS' effectiveness in finding adversarial samples. This is mainly done by leveraging PANTS to find adversarial samples against vanilla models under different applications. Then, we evaluate the ASR to find adversarial samples against models robustified by different methods such as PANTS-robustified, Amoeba-robustified and NetShare-robustified ones. We show that PANTS can robustify MNCs more than Amoeba and NetShare cannot robustify the MNCs. (E1) demonstrates our major claim of (C1), (C3) and (C5).

How to: Reviewers can run the following two scripts sequentially to evaluate (E1)

```
$ cd PANTS/scripts
$ bash test-asr.sh
$ bash test-netshare.sh
```

Results: The majority of results are directly plotted as figures in `PANTS/figures`. After the two scripts finish, there should be multiple figures generated to validate (C1), (C3) and C(5). The first batch are

- `app_end_host_vanilla.pdf`
- `app_in_path_vanilla.pdf`
- `vpn_end_host_vanilla.pdf`
- `vpn_in_path_vanilla.pdf`
- `vca_end_host_vanilla.pdf`
- `vca_in_path_vanilla.pdf`

They show the ASR when using PANTS to against vanilla models under different applications and threat models. The corresponding figure in the paper is Fig. 6. The second batch shows the ASR on using PANTS against PANTS-robustified, Amoeba-robustified, and NetShare-robustified models. They are

- `app_end_host_robustified.pdf`
- `app_in_path_robustified.pdf`
- `vpn_end_host_robustified.pdf`
- `vpn_in_path_robustified.pdf`
- `vca_end_host_robustified.pdf`
- `vca_in_path_robustified.pdf`
- `netshare_mlp.pdf`
- `netshare_rf.pdf`
- `netshare_tf.pdf`
- `netshare_cnn.pdf`

Their corresponding figures and table are Fig. 9, Fig. 10 and Table 2 in the paper.

Finally, the generation speed for Fig. 12 is logged in `PANTS/logs/{app,vpn,vca}/..._vanilla/results.txt`.

(E2): [1 human-minute + 9 compute-hour + 20GB disk]: (E2) also consists of two parts. First, it evaluates the model accuracy and robustness for multiple robustification approaches. Then, it shows the improvement of ASR when using PANTS' generated samples for iteratively robustifying the models. (E2) demonstrates our major claim of (C2).

How to: Users are required to run two scripts sequentially

```
$ cd PANTS/scripts
$ bash test-adv-train.sh
$ bash test-robustification.sh
```

Results: The script plots figures

- `adv_train.pdf`
- `robustfication_end-host.pdf`

Please refer to Fig. 7 and Fig. 8 in the original paper for validation.

(E3): [1 human-minute + 11 compute-hour + 20GB disk]: (E3) evaluates the PANTS-robustified models against threat models outside those used during robustification. It mainly validates the major claim C4.

How to: Users should run one script

```
$ cd PANTS/scripts
$ bash test-more-threat.sh
```

Results: The script plots the figure, `threat.pdf`. It corresponds to Fig. 11.

(E4): [1 human-minute + 1 compute-hour + 20GB disk]: (E4) mainly evaluates the transferability for the samples generated by PANTS. It corresponds to the major claim (C6).

How to: Users are required to run one script

```
$ cd PANTS/scripts
$ bash test-transferability.sh
```

Results: The script plots the figure, `transferability.pdf`. Its corresponding figure is Fig. 13.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: A Crack in the Bark: Leveraging Public Knowledge to Remove Tree-Ring Watermarks

Junhua Lin Marc Juarez
University of Edinburgh

A Artifact Appendix

A.1 Abstract

We provide the source code used in our experiments which implements the set of attacks that successfully remove a Tree-Ring Watermark. The primary contribution is an implementation of projected gradient descent using a surrogate classifier which operates in the latent space of the diffusion model. Currently our attacks only operate within the latent space provided by Stable Diffusion v2.1, SDXL and Ostris (a 16-Channel VAE). We also provide the implementation of another attack called Adversarial noising, created by Nils et al., which was adapted to our evaluation framework. Additionally, we provide the images used as the target class images which our attacks attempt to imitate. This is given as two sets, one set (named wmvsunwm_imagenet) contains 1500 images to be used as reference images in assessing Wm vs Unwm attacks and the other set (named wmvspub_imagenet) contains 7000 images to be used in Wm vs Public attacks. Both sets of images have their corresponding set of prompts inside the prompts directory of the repository.

A.2 Description & Requirements

Within the repository, we include a README file to describe the Python environment used to produce the results reported in our paper. Additionally, we also provide a DockerFile to allow users to build a virtual machine with all the necessary libraries. Beyond the libraries we also require the user to install model weights that need to be placed within the base directory of the repository. These are OpenAI's imagenet diffusion weights [1] and Alexnet weights [2].

A.2.1 Security, privacy, and ethical concerns

Our attacks specifically target images and does not affect the system in any way. Furthermore, the datasets and models used are open sourced by their respective authors. As such, there is no risk associated with the security of the machine or data privacy. With regard to ethical concerns, Tree-Ring watermarking is still in the prototype stage and has not been deployed. Therefore, the likelihood of malicious actors exploiting these attacks is minimal, while exposing their vulnerabilities gives a better understanding of the limitations of ML watermarking.

A.2.2 How to access

A stable version of the artifact is provided in Zenodo [3]. The stable URL points to the latest version of the artifact which contains three archives. These are `wmvspub_imagenet.tar.gz` which contains 7000 images sampled from ImageNet, `wmvsunwm_imagenet.tar.gz` which contains 1500 images from ImageNet and `crack-in-the-bark-main.zip` which contains the source code for our experiments.

A.2.3 Hardware dependencies

Our experiments were run on an Nvidia A100 GPU with 80GB of VRAM with 8 CPU cores. The minimum memory requirement is 32GB with atleast 50GB of storage available.

A.2.4 Software dependencies

All software dependencies are listed in the README file inside the repository. The minimum requirements are Python and a package manager (e.g. conda/pip/mamba). The Dockerfile also constructs a machine image with the Python environment already setup.

A.2.5 Benchmarks

The datasets are provided within the Zenodo repository for this project. The diffusion models such as Stable Diffusion, SDXL's VAE of Ostris's VAE are installed when the code is run. Additional models such as Alexnet's weights and Guided Diffusion's weights were detailed in Section A.2.

[1] https://openaipublic.blob.core.windows.net/diffusion/jul-2021/512x512_diffusion.pt

[2] https://github.com/richzhang/PerceptualSimilarity/tree/master/lpips/weights

[3] https://doi.org/10.5281/zenodo.15595719

A.3 Set-up

A.3.1 Installation

Installation of packages should be done using a python package manager (e.g. conda/pip/mamba) if not using the Docker image. The version of Python we use is 3.10 and the CUDA drivers are version 12.6. After unzipping the source code, you will need to install and unzip the imagenet datasets from Zenodo into the base directory of the repository. You will also need to clone the Perceptual Similarity repository and move the "weights" directory inside the "lpips" directory to the base of the repository. Finally, you will need to download the Guided Diffusion weights from OpenAI and place them within the base of the repository.

Note that the version of `huggingface_hub` used has a bug related to the version of the `diffusers` library used. As such, we require the user to manually edit the `dynamic_modules_utils.py` file. Details on what to edit and where to find the file are described in the source code's README under the section 'Dependencies'. This step can be skipped if

A.3.2 Basic Test

To verify that the environment has been properly set up, we provide a simple verification experiment via 'verify.sh'. Running this script will start a pipeline which goes through each stage of the attack. This simple experiment should take no longer than 10 minutes, depending on the GPU used. After this script has finished running, a log directory will be made containing two folders, one containing all the data, i.e. the cached latents, images, prompt to image tables e.t.c. The other containing the evaluation results. Additionally, an output folder should have been created with two subdirectories, 'images' and 'models'. The should contain the output images of the attack and the surrogate model used to execute the attack respectively.

A.4 Evaluation workflow

A.4.1 Major Claims

The main claims made in our paper are:

(C1): All the surrogate detectors (trained on Raw Pixel Values, True Latent Vectors, and VAE-Recovered Latent Vectors) achieve high accuracy and none of them significantly overfits. This is shown by experiments (E3)–(E6), and (E11) and (E12) described in Section 5.2 whose results are reported in Table 1.

(C2): Our evaluation of adversarial noising achieves a TPR@1%FPR of 0.141 for $\delta = 2$, slightly higher than the reported 0.052 of the original paper. This is proven by experiment (E2) described in Section 5.3 and reported in the second row of Table 2.

(C3): We found a discrepancy between An et al.'s and our results when executing the PGD attack with a surrogate trained on the Raw Pixel Values of the Wm & Pub dataset: they reported an average TPR@1%FPR of 0.99, but our results show a value of 0.48. This is proven by experiments (E3) and (E4) described in Section 5.3 and reported in the third row of Table 2.

(C4): Our PGD attack on the True Latents Vectors (fourth row of Table 2) performs worse than our surrogate attack trained on VAE-Recovered Latent Vectors. This is supported by experiments (E5), (E6), (E11), and (E12) described in Section 5.3 and reported in rows 5 and 6 of Table 2.

(C5): Among the attacks we evaluated, our proposed attack by recovering latents using AutoencoderKL (the same autoencoder as the target) results in the largest drop in accuracy, while maintaining image quality comparable to the baseline. This is supported by experiments (E2)–(E6), and (E11) and (E12), described in Section 5.3 and reported in Table 2.

(C6: All the attacks included in our evaluation are significantly more effective than any of the approaches tested by Wen et al.: they report an average ROC-AUC of 0.975 and TPR@1%FPR of 0.694, while the attacks we evaluate achieve an average of 0.519 ROC-AUC and 0.261 TPR@1%FPR, representing a 1.9x decrease in overall detection accuracy. Our results are obtained by conducting experiments (E2)–(E6), and (E11) and (E12), described in Section 5.3 and reported in Table 2.

(C7): Using SDXL's VAE, the attack remains robust, though less effective than when using the victim model's VAE and with lower image quality. This is supported by experiments (E7) and (E8) described in Section 5.4 and reported in row 5 of Table 2.

(C8): Training the attack with the SDXL's VAE on the Wm & UnWm dataset significantly degrades image quality, with the FID score increasing by 33.71 and similar increases in LPIPS and CLIP Score. This is demonstrated by experiments (E7) and (E8) described in Section 5.4 and shown in row 5 of Table 2.

(C9): The 16-Channel VAE's performance is comparable to training the surrogate detector on raw images. This is shown by experiments (E9) and (E10) described in Section 5.4 and shown in row 6 of Table 2.

(C10): While the attack still decreases the effectiveness of Tree-Ring, it is not as successful as when the target VAE is available. This is supported by experiments (E13) and (E14) described in Section 5.5 and reported in Table 3.

(C11): In the no-attack setting, for base rates below 0.1, precision decreases significantly, even for high TPRs. This is supported by experiments (E1) and (E15) described in Section 5.6 and illustrated in the no-attack subplot of Figure 8.

A.4.2 Experiments

Within the repository, a detailed README is provided, called ‘example_pipelines.md‘. After the setup instructions provided in Section A.3.1, all thats needed is to follow the example pipelines provided whilst substituting the names of directories automatically generated after completion of certain scripts.

(E1): No Attack Baseline [20 human-minutes + 12 compute-hours + 1GB disk]: Generate watermarked images and assess detection performance without any attack to establish baseline metrics.
How to: Follow the "No Attack" pipeline in example_pipelines.md. Run generate_data.py then assess_images.py using the same watermarked images as both original and adversarial inputs to establish baseline detection rates.
Results: Baseline watermark detection performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in summary.json under the assessment logs directory. If no changes were made to the --run_name parameter, the directory created will have "no_attack" in its name.

(E2): Adversarial Noising Attack [20 human-minutes + 24 compute-hours + 2GB disk]: Reproduce the baseline adversarial noising attack from prior work for comparison with our methods.
How to: Follow the "Adversarial Noising" pipeline in example_pipelines.md. Generate watermarked data, then run remove_watermark.py with --attack=adv_noising and --mode=rawpix, followed by assessment.
Results: Adversarial Noising performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in summary.json under the assessment logs directory. If no changes were made to the --run_name parameter, the directory created will have "adv_noising" in its name.

(E3): Raw Pixel Surrogate Attack - WM vs UnWM [20 human-minutes + 8 compute-hours + 2GB disk]: Train surrogate model on raw pixels comparing watermarked vs generated non-watermarked images, then perform PGD-based removal.
How to: Follow the "Raw Pixel Values - Wm vs UnWm" pipeline in example_pipelines.md. Generate both watermarked and non-watermarked images, train surrogate with --mode=rawpix, then perform removal and assessment.
Results: Raw Pixel Wm vs UnWm performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in summary.json under the assessment logs directory. If no changes were made to the --run_name parameter, the directory created will have "wmvsunwm_rawpix" in its name.

(E4): Raw Pixel Surrogate Attack - WM vs Public [20 human-minutes + 16 compute-hours + 8GB disk]: Train surrogate model comparing watermarked images vs public dataset images for more realistic attack scenario.
How to: Follow the "Raw Pixel Values - Wm vs Public" pipeline in example_pipelines.md. Generate watermarked images, train surrogate against wmvspub_imagenet dataset with --mode=rawpix, then perform removal and assessment.
Results: Raw Pixel Wm vs Public performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in summary.json under the assessment logs directory. If no changes were made to the --run_name parameter, the directory created will have "wmvspub_rawpix" in its name.

(E5): True Latent Vector Attack - WM vs UnWM [20 human-minutes + 8 compute-hours + 2GB disk]: Attack using true latent vectors from diffusion process comparing watermarked images vs generated non-watermarked images.
How to: Follow the "True Latent Vectors - Wm vs UnWm" pipeline in example_pipelines.md. Generate data with --save_raw_latent flag when running generate_data.py, train surrogate, perform latent-space removal and assess.
Results: True Latent Wm vs UnWm performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in summary.json under the assessment logs directory. If no changes were made to the --run_name parameter, the directory created will have "wmvsunwm_true_latent" in its name.

(E6): True Latent Vector Attack - WM vs Public [20 human-minutes + 16 compute-hours + 8GB disk]: Attack using true latent vectors from diffusion process comparing watermarked images vs public images.
How to: Follow the "True Latent Vectors - Wm vs Public" pipeline in example_pipelines.md. Generate watermarked data with --save_raw_latent, recover latents for public images using recover_latents.py, train, attack and assess.
Results: True Latent Wm vs Public performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in summary.json under the assessment logs directory. If no changes were made to the --run_name parameter, the directory created will have "wmvspub_true_latent" in its name.

(E7): SDXL-VAE Latent Attack - WM vs UnWM [20 human-minutes + 8 compute-hours + 2GB disk]: Attack using SDXL VAE-recovered latents against generated

non-watermarked dataset.

How to: Follow the "SDXL-VAE Latent Vectors - Wm vs UnWm" pipeline in `example_pipelines.md`. Generate images, recover latents using SDXL VAE (`stabilityai/sdxl-vae`), train surrogate, perform removal and assess images

Results: SDXL-VAE Wm vs UnWm performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in `summary.json` under the assessment logs directory. If no changes were made to the `--run_name` parameter, the directory created will have "wmvsunwm_sdxl_latent" in its name.

(E8): SDXL-VAE Latent Attack - WM vs Public [20 human-minutes + 16 compute-hours + 8GB disk]: Attack using SDXL VAE-recovered latents against public dataset.

How to: Follow the "SDXL-VAE Latent Vectors - Wm vs Public" pipeline in `example_pipelines.md`. Generate watermarked images, recover SDXL latents for both watermarked and public images, train surrogate and perform attack.

Results: SDXL-VAE Wm vs Public performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in `summary.json` under the assessment logs directory. If no changes were made to the `--run_name` parameter, the directory created will have "wmvsunwm_sdxl_latent" in its name.

(E9): 16-Channel VAE Attack - WM vs UnWM [20 human-minutes + 8 compute-hours + 2GB disk]: Attack using high-dimensional VAE latents with 16 channels against generated non-watermarked dataset.

How to: Follow the "16-Channel VAE Latent Vectors - Wm vs UnWm" pipeline in `example_pipelines.md`. Use `ostris/vae-kl-f8-d16` model for latent recovery, train surrogate in high-dimensional latent space, perform the attack and assess.

Results: 16-Channel VAE Wm vs UnWm performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in `summary.json` under the assessment logs directory. If no changes were made to the `--run_name` parameter, the directory created will have "wmvsunwm_ostris_latent" in its name.

(E10): 16-Channel VAE Attack - WM vs Public [20 human-minutes + 16 compute-hours + 8GB disk]: Attack using high-dimensional VAE latents with 16 channels against public dataset.

How to: Follow the "16-Channel VAE Latent Vectors - Wm vs Public" pipeline in `example_pipelines.md`. Recover 16-channel latents for both watermarked and public images using `ostris/vae-kl-f8-d16`, then train, attack and assess.

Results: 16-Channel VAE Wm vs Public performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in `summary.json` under the assessment logs directory. If no changes were made to the `--run_name` parameter, the directory created will have "wmvspub_ostris_latent" in its name.

(E11): VAE - Recovered Attack - WM vs UnWM [20 human-minutes + 8 compute-hours + 2GB disk]: Attack using SD VAE-recovered latents against generated non-watermarked dataset.

How to: Follow the "VAE-Recovered Latent Vectors - Wm vs UnWm" pipeline in `example_pipelines.md`. Use `stabilityai/stable-diffusion-2-1-base` for latent recovery, train surrogate with, perform attack and assess.

Results: SD VAE Wm vs UnWm performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in `summary.json` under the assessment logs directory. If no changes were made to the `--run_name` parameter, the directory created will have "wmvsunwm_sd_latent" in its name.

(E12): VAE - Recovered Attack - WM vs Public [20 human-minutes + 16 compute-hours + 8GB disk]: Attack using SD VAE-recovered latents against public dataset.

How to: Follow the "VAE-Recovered Latent Vectors - Wm vs Public" pipeline in `example_pipelines.md`. Recover SD VAE latents for watermarked images and public dataset, train surrogate, perform attack and assess.

Results: SD VAE Wm vs Public performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in `summary.json` under the assessment logs directory. If no changes were made to the `--run_name` parameter, the directory created will have "wmvspub_sd_latent" in its name.

(E13): Guided Diffusion Attack - WM vs UnWM [20 human-minutes + 30 compute-hours + 6GB disk]: Attack using OpenAI's Guided Diffusion model comparing against generated non-watermarked dataset.

How to: Follow the "Attack on Guided Diffusion Model - Wm vs UnWm" pipeline in `example_pipelines.md`. Use `--model_id=512x512_diffusion` for generation assuming you have placed the model at the base directory after installing from the Guided Diffusion repository, train surrogate, perform attack and assess.

Results: Guided Diffusion Wm vs UnWm performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in `summary.json` under the assessment logs directory. If no changes were made to the `--run_name` parameter, the directory created will have "wmvsunwm_guided_diffusion" in its name.

(E14): Guided Diffusion Attack - WM vs Public [20 human-minutes + 48 compute-hours + 8GB disk]: Attack using OpenAI's Guided Diffusion model comparing against public dataset.

How to: Follow the "Attack on Guided Diffusion Model - Wm vs Public" pipeline in `example_pipelines.md`. Generate larger dataset `--max_num_images=7000` with Guided Diffusion, train against public images, perform attack, and assess attack performance.

Results: Guided Diffusion Wm vs Public performance metrics (PRC-AUC, ROC-AUC, Accuracy, TPR@1%FPR) and image quality scores (LPIPS, CLIP, FID) stored in `summary.json` under the assessment logs directory. If no changes were made to the `--run_name` parameter, the directory created will have "wmvspub_guided_diffusion" in its name.

(E15:) Precision vs Base Rate [20 human-minutes + 1GB disk]: Plotting the precision at different TPR@X%FPR cutoffs against several base rates.

How to: Follow the "Precision vs Base Rate Plot" example in `example_pipelines.md`. Call `precision_at_base_rate.py` with `--table_path` set to the path to a `metadata.csv` file after `assess_images.py` completes.

Setup: The assumption here is that you have atleast one experiment where `assess_images.py` has completed. After which, the logger object will create a csv file under the experiment results directory in the logs directory which contains the measurement metric for deciding the validity of a watermark.

Results: This will generate a single plot for an attack where we selected 4 TPR@X%FPR cut off points and plotted the precision against a set of base rates.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix - CertTA: Certified Robustness Made Practical for Learning-Based Traffic Analysis

Jinzhu Yan[1] Zhuotao Liu[1,2,✉] Yuyang Xie[1] Shiyu Liang[3] Lin Liu[4] Ke Xu[1,2]
[1] Tsinghua University [2] Zhongguancun Laboratory
[3] Shanghai Jiao Tong University [4] National University of Defense Technology

A Artifact Appendix

A.1 Abstract

This appendix contains research artifacts associated with our USENIX Security'25 paper CertTA, which presents the first solution that provides certifiable robustness against multi-modal adversarial attacks in traffic analysis models. We implement a prototype of CertTA and extensively evaluate it against three categories of multi-modal adversarial attacks across six traffic analysis models and two datasets. Our experimental results demonstrate that CertTA provides significantly stronger robustness guarantees than the state-of-the-art approaches when confronting adversarial attacks.

In this appendix, we present a repository containing two traffic datasets, three attack methodologies to generate adversarial flows, six supervised traffic analysis models, CertTA and three baseline certification methods to construct certified traffic analysis models, three unsupervised anomaly detectors and a novel integrated system that synergistically combines certified traffic analysis models with anomaly detectors. In order to facilitate the reproduction of our key experimental results and foster future research, we also provide step-by-step instructions for running our CertTA prototype and a step-by-step demo of key experiments in the repository.

A.2 Description & Requirements

Our artifacts are organized into the following directories:

- **dataset/** contains the CICDOH20 and TIISSRC23 datasets, including the processed json files and original PCAP files of flow samples.
- **model/** contains the implementations of six supervised traffic analysis systems (*i.e.,* kFP, Kitsune (supervised), Whisper (supervised), DFNet, YaTC and TrafficFormer) and three unsupervised anomaly detection systems (*i.e.,* KMeans, Kitsune, Whisper).
- **certification/** contains the implementations of CertTA's multi-modal smoothing mechanism and the functions for solving CertTA's robustness region against multi-modal adversarial perturbations.
- **attack/** contains the implementations of three multi-modal adversarial attacks (*i.e.,* Blanket, Amoeba, Prism).
- **BARS/** contains the implementations of a baseline certification method BARS.
- **evaluation/** contains the source codes for training and evaluating certified traffic analysis models. Our prototype supports both CertTA and baseline certification methods (*i.e.,* VRS, BARS and RS-Del) for building certified traffic analysis models.
- **integration/** contains the source codes for building and evaluating the integrated system of anomaly detectors and certified traffic analysis models.

A.2.1 Security, Privacy, and Ethical Concerns

Our artifacts are non-destructive, and there is no security or privacy risk when running the CertTA prototype. The datasets used in our evaluations are publicly available, and all third-party artifacts are based on open-source implementations. We strictly followed all terms of use, and no private or sensitive data were accessed or disclosed.

A.2.2 How to Access

Our repository is archived on Zenodo[1], with a GitHub[2] mirror for ease of access. The research artifacts, including the source code of our CertTA prototype and the experimental artifacts (*e.g.,* the datasets, the detailed implementations of traffic analysis models, adversarial attack methodologies and baseline approaches), can be accessed via these public repositories under an open-source license.

A.2.3 Hardware Dependencies

As a reference, our experiments are conducted on a Supermicro SYS-740GP-TNRT server with two Intel(R) Xeon(R) Gold 6348 CPUs (2 × 28 cores), 512GB RAM, one NVIDIA

✉ Corresponding author.

[1] Available at https://doi.org/10.5281/zenodo.15580292
[2] Available at https://github.com/InspiringGroup-Lab/CertTA

A100 GPU and two NVIDIA GeForce RTX 4090 GPUs. Since we have not verified the compatibility of our artifacts in a CPU-only hardware environment, we recommend conducting experiments on a machine equipped with GPUs.

A.2.4 Software Dependencies

As a reference, our artifacts have been successfully tested in Ubuntu 20.04 server with Python 3.8.18. Based on the Anaconda distribution, we provide step-by-step instructions about software environment setup in our repository. All required Python packages are listed in the "environment.yml" file.

A.2.5 Benchmarks

The CICDOH20 and TIISSRC23 traffic datasets, the pre-trained checkpoints of traffic analysis models YaTC and TrafficFormer are required by the experiments with our artifacts. These datasets and model checkpoints are completely provided in our Zenodo repository, while the PCAP files of the two traffic datasets and the pre-trained checkpoint of the TrafficFormer model are not provided in our Github repository due to the repository size limit.

A.3 Set-up

A.3.1 Installation

1. Download the zip file of our Zenodo repository and unzip the complete artifacts.
2. Ensure that you have *conda* installed on your system. If you do not have *conda*, you can install it as part of the Anaconda distribution or Miniconda.
3. Open a terminal or command prompt.
4. Create a new *conda* environment with the name of your choice (*e.g.,* CertTA) and install all the required packages listed in the "environment.yml" file:

```
conda create -n CertTA -f environment.yml
```

5. Once the environment is created, activate it by running:

```
conda activate CertTA
```

6. Switch to the root directory of the repository (*i.e.,* the *CertTA_public* directory).

A.3.2 Basic Test

To check that all required software components are used and functioning fine, run the command:

```
python evaluation/setup_check.py
```

The evaluation environment is properly initialized if "Setup Successful!" appears at the end of the output.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): CertTA provides much stronger robustness guarantees against multi-modal adversarial attacks than the SOTA approaches (*i.e.,* VRS, BARS and RS-Del), while imposing very minimal performance reductions on clean traffic. This is proven by the experiment (E1) described in Section 5.2 whose results are illustrated in Table 5, Figure 6 and Figure 7.

(C2): The synergistic integration between CertTA and anomaly detection systems can create a fundamental dilemma for the attacker, thereby achieving consistently high Defense Success Rate against adversarial attacks with varying attack intensities. This is proven by the experiment (E2) described in Section 5.3 whose results are illustrated in Figure 8.

A.4.2 Experiments

Running all the experiments in our paper - involving two traffic datasets, three adversarial attack methodologies, six supervised traffic analysis systems, CertTA and three baseline certification methods, three unsupervised anomaly detectors and an integrated system - might take several weeks to complete. To facilitate a quick verification of the major claims made in our paper, we provide a step-by-step demo in the repository to reproduce key experiments. In this demo, we use the CICDOH20 dataset and the YaTC model to build CertTA-certified/baseline traffic analysis models and reproduce key experimental results based on these models.

(E1): [Certified Robustness against Adversarial Attacks] [4 human-hours + 6 compute-hours]: Using the CICDOH20 dataset and the YaTC model, we reproduce key experimental results in Table 5 and Figure 6 to verify the major claim (C1).

Step 1: Train CertTA-certified/baseline traffic analysis models. The trained model checkpoints and training logs will be saved. To save time, we provide pre-saved model checkpoints to proceed with subsequent experiment steps.

Step 2: Evaluate the performance on clean traffic. We evaluate the classification performance of CertTA-certified/baseline traffic analysis models on clean traffic. The accuracy/precision/recall/F_1-score of each traffic class and their macro aggregation will be saved, which can be compared with the results in Table 5.

Step 3: Measure the robustness region of the CertTA-certified traffic analysis model. Based on the robustness region offered by CertTA-certified traffic analysis model, we plot the CDF curves of certified accuracy under different robustness radius. The plotted figure will be saved, which can be compared with the sub-figure on the left-most column of Figure 6.

Step 4: Generate adversarial flows. We train the Blanket, Amoeba and Prism attack models to generate adversarial flows. The trained model checkpoints, training logs and generated adversarial flows will be saved. To save time, we provide pre-saved datasets of these adversarial flows to proceed with subsequent experiment steps.

Step 5: Evaluate the certified accuracy against adversarial flows. We evaluate the certified accuracy of CertTA-certified/baseline traffic analysis models against adversarial flows. The certified accuracy of each traffic class and their macro aggregation will be saved, which can be compared with the results in the right-side three columns of Figure 6.

Step 6: Verify the reproduced experimental results.

(E2): [Integration with Anomaly Detection] [3 human-hours + 5 compute-hours]: Using the CICDOH20 dataset and the YaTC model, we reproduce key experimental results in Figure 8 to verify the major claim (C2).

Step 1: Generate adversarial flows of different intensities. We train the Blanket, Amoeba and Prism attack models to generate adversarial flows with different levels of attack intensities. The trained model checkpoints, training logs and generated adversarial flows will be saved. To save time, we provide pre-saved datasets of these adversarial flows to proceed with subsequent experiment steps.

Step 2: Evaluate the non-certified traffic analysis model against adversarial flows. The accuracy of each traffic class and their macro aggregation will be saved, which can be compared with the results in the left two columns of Figure 8.

Step 3: Train the anomaly detector. We train the Kitsune model for anomaly detection against adversarial flows. The trained model checkpoint and training log will be saved.

Step 4: Evaluate the standalone anomaly detector against adversarial flows. The False Positive Rate on clean traffic and the True Positive Rate on adversarial flows will be saved, which can be compared with the results in the middle two columns of Figure 8.

Step 5: Evaluate the standalone certified traffic analysis model against adversarial flows. The accuracy of each traffic class and their macro aggregation will be saved, which can be compared with the results in the right two columns of Figure 8.

Step 6: Evaluate the integrated system against adversarial flows. The Defense Success Rate on each traffic class and their macro aggregation will be saved, which can be compared with the results in the middle two columns and right two columns of Figure 8.

Step 7: Verify the reproduced experimental results.

The commands to be executed in each experiment step and the organization of the experimental results are detailed in the Demo.md file. All intermediate model checkpoints and experimental results are saved to enable easy comparisons with the results found in our paper. Beyond this demo, you can also follow the aforementioned experiment steps to reproduce results using other datasets and models.

A.5 Notes on Reusability

Our artifacts integrate two traffic datasets, six traffic analysis models and three anomaly detectors using different flow representations (*e.g.,* flow statistics, raw flow sequences and raw bytes) and architectures (*e.g.,* traditional machine learning based, deep learning based and Transformer based), three categories of adversarial attacks (*e.g.,* Generative Adversarial Network based, Reinforcement Learning based and explicit modeling based) and four robustness certification methods. Based on the step-by-step instructions in our repository, these artifacts can be reused to facilitate future studies and deployment of traffic analysis systems.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Invisible but Detected: Physical Adversarial Shadow Attack and Defense on LiDAR Object Detection

Ryunosuke Kobayashi[1], Kazuki Nomoto[1,2], Yuna Tanaka[1], Go Tsuruoka[1], Tatsuya Mori[1,3,4]
[1]Waseda University, [2]Deloitte Tohmatsu Cyber LLC, [3]NICT, [4]RIKEN AIP

A Artifact Appendix

A.1 Abstract

This artifact enables the reproduction of all experiments conducted in our paper. To facilitate experiment replication, we provide the code and data at https://zenodo.org/records/15120571. All experiments are based on OpenPCDet and AWSIM, which are publicly available open-source software. By setting up these environments and downloading our code and data, you can reproduce the experiments. We hope this artifact will be useful for your research.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There is no risk for users regarding their machine's security, data privacy, or other ethical concerns while executing our artifact.

A.2.2 How to access

The data and code used in our experiments are available at https://zenodo.org/records/15120571. However, the provided code runs on OpenPCDet and AWSIM, both of which are open-source software. The version of OpenPCDet used in our experiments is available at https://github.com/kbys0519/OpenPCDet.git. Similarly, AWSIM can be accessed at https://github.com/kbys0519/AWSIM.git.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Our experiment consists of the following three steps:
Step 1: Optimization of the Adversarial Shadow (Section 4.3)
Step 2: Reproduction of the Adversarial Shadow in the simulator and point cloud data collection (Section 6)
Step 3: Output of object detection results from point cloud data (Sections 6, 7)

To reproduce our experiment, there are two options:

Option 1: Verify object detection results using the provided data (reproduce only Step 3)
Option 2: Reproduce the entire process, including optimization, point cloud data collection, and object detection results (reproduce Steps 1, 2, and 3)

For Option 1, setting up OpenPCDet is sufficient, as the simulator is not required. For Option 2, in addition to OpenPCDet, the autonomous driving simulator AWSIM must also be set up.

The software requirements for each option are listed below:
For Option 1.

- Ubuntu 20.04/22.04
- Python 3.6+
- PyTorch 1.1 or higher
- CUDA 9.0 or higher
- spconv v1.0 (commit 8da6f96) or spconv v1.2 or spconv v2.x

For Option 2.

Please prepare two PCs: one with the Option 1 setup and another with the following setup. We have not verified the operation of OpenPCDet and AWSIM on the same PC.

- Ubuntu 22.04
- ROS2 Humble
- Unity 2021.1.7f1
- Vulkan Graphics Library

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

For experiments with **Option 1**, prepare a PC 1 with Ubuntu 20.04 or Ubuntu 22.04. For experiments with **Option 2**, in addition to the **Option 1** setup, prepare an additional PC 2

with Ubuntu 22.04. Install the required libraries on each PC as described in Section A.2.4. The required installations for the experiment are listed below.

PC 1.

Clone the OpenPCDet repository.

```
$ git clone git@github.com:kbys0519/OpenPCDet.git
```

Move to the directory.

```
$ cd OpenPCDet
```

Install pcdet library.

```
$ python3 setup.py develop
```

Install Open3D.

```
$ pip3 install open3d
```

Finally, download the models from the OpenPCDet repository https://github.com/open-mmlab/OpenPCDet/tree/master?tab=readme-ov-file#model-zoo.

PC 2.

Clone the AWSIM repository.

```
$ git clone git@github.com:kbys0519/AWSIM.git
```

Launch Unity Hub and open AWSIM in the Unity Hub.

Download the map files from https://github.com/tier4/AWSIM/releases/download/v1.2.2/Japan_Tokyo_Nishishinjuku_v2.unitypackage

In the AWSIM project, import the package.

```
Assets/Import Package/Custom Package
```

A.3.2 Basic Test

The artifact's data includes the point cloud data used in the experiments for each section. Download these data and perform an object detection test using PC 1 only.

Move to the directory.

```
$ cd OpenPCDet/tools
```

Run the inference.

```
$ python3 demo.py -cfg_file
cfgs/kitti_models/[MODEL NAME].yaml
-ckpt [DOWNLOADED MODEL] -data_path
[POINT_CLOUD_DATA]
```

Then, the object detection results will be displayed.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Using a point cloud dataset with no surrounding objects, the optimal Shadow Shape to deceive the object detection model can be optimized, as shown in Figure 5 of our paper.*

(C2): *Our defense method against Shadow Hack, BBValidator, can achieve a defense success rate for specific parameters, as shown in Figure 21 of our paper.*

A.4.2 Experiments

(E1): *[Optimizing Shadow Shape] [15 human-minutes + 30 compute-minutes]: Search for the optimal shadow shape for the attack using point clouds in an environment with no surrounding objects. The output consists of the scores for each shape.*

Preparation: *Download flat/0.bin. This is a point cloud data collected in an empty environment for optimization purposes. Then, download optimize.py and move it to OpenPCDet/tools.*

Execution: *Replace demo.py in Section A.3.2 with optimize.py, replace the point cloud with flat/0.bin, and then execute the script.*

Results: *The output consists of the scores for each shape.*

(E2): *[Defense Method] [5 human-minutes + 1 compute-minutes]: By replacing the standard demo.py with defense.py, it is possible to evaluate the defense method.*

Preparation: *Download defense.py.*

Execution: *Replace demo.py in Section A.3.2 with defense.py. Then, perform inference on each point cloud dataset.*

Results: *It can be confirmed that the Shadow Hack attack is successfully prevented.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix - From Threat to Trust: Exploiting Attention Mechanisms for Attacks and Defenses in Cooperative Perception

Chenyi Wang[1] Raymond Muller[2] Ruoyu Song[2] Jean-Philippe Monteuuis[3]
Jonathan Petit[3] Yanmao Man[5] Ryan Gerdes[4] Z. Berkay Celik[2] Ming Li[1]
[1]University of Arizona [2]Purdue University [3]Qualcomm [4]Virginia Tech [5]Independent Researcher, U.S.
{chenyiw, yman, lim}@arizona.edu, {mullerr, song464, zcelik}@purdue.edu
{jmonteuu, petit}@qti.qualcomm.com, rgerdes@vt.edu

A Artifact Appendix

A.1 Abstract

This artifact provides the implementation and evaluation framework for the paper accepted at USENIX Security 2025 Cycle 2 - From Threat to Trust: Exploiting Attention Mechanisms for Attacks and Defenses in Cooperative Perception, which introduces SOMBRA, a novel cooperative perception attack, and LUCIA, a novel light-weight defense. The artifact includes code for generating adversarial perturbations, using both SOMBRA and the prior art method, evaluating their impact on state-of-the-art cooperative perception models. The codebase also includes the implementation for the defense LUCIA, compared with the official ROBOSAC implementation. The artifact also includes a case study on traffic jam scenarios. The provided instructions enable evaluators to reproduce the results and validate the claims made in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no known security, privacy, or ethical concerns associated with running this artifact. The artifact does not require disabling security mechanisms or accessing sensitive data. All datasets used are publicly available and properly cited.

A.2.2 How to access

The artifact is hosted on GitHub and can be accessed at `https://github.com/WiSeR-Lab/SOMBRA_LUCIA`*. A stable version of the artifact, including instructions and the collected custom dataset, is archived on Zenodo at* `https://doi.org/10.5281/zenodo.15523768` *and* `https://zenodo.org/records/15523769`*, respectively.*

A.2.3 Hardware dependencies

The artifact requires a machine with the following specifications:

- GPU: NVIDIA GPU with CUDA support (tested on CUDA 11.8 and 12.0).
- Minimum 8 GB of GPU memory.
- Minimum 16 GB of system memory.
- Disk space: At least 50 GB for datasets, models, and intermediate results.

A.2.4 Software dependencies

The artifact requires the following software:

- Operating System: Linux (tested on Ubuntu 20.04 and 22.04).
- Python: Version 3.8 or higher.
- Package manager: `pixi` for environment setup (`https://pixi.sh`).
- PyTorch: Version compatible with the CUDA version (e.g., PyTorch with CUDA 11.8 or 12.0).
- Additional Python packages: Listed in the `pixi.toml` file included in the repository.

A.2.5 Benchmarks

The artifact uses the following datasets and models:

- Dataset: OPV2V test split, available at https://mobility-lab.seas.ucla.edu/opv2v/.
- Pretrained models: Attentive Fusion, CoAlign, Where2comm, and V2VAM, available at https://github.com/DerrickXuNu/OpenCOOD/tree/main.
- Traffic jam case study dataset: Available at https://zenodo.org/records/15523769. A data and model downloading script is provided with the artifact, with instructions given in A.3.3.

A.3 Set-up

A.3.1 Installation

Follow these steps to set up the environment:
1. Install `pixi` by following the instructions at https://pixi.sh.
2. Clone the artifact repository:

```
git clone \
https://github.com/WiSeR-Lab/SOMBRA_LUCIA.git
cd SOMBRA_LUCIA
```

3. Activate the environment and install dependencies:

```
# One-liner setting up and activating VE
pixi shell
# Setup opencood package
python setup.py develop
# Build IoU operation on CUDA
python opencood/utils/setup.py build_ext --inplace
```

A.3.2 Basic Test

Run the following command to verify the setup:

```
python cp_attack.py --help
```

Expected output: A list of available arguments and their descriptions. This confirms that the environment is correctly set up.

A.3.3 Dataset and Model Weights

We use the test split of the OPV2V dataset, and the official pretrained weights for Attentive Fusion, CoAlign, Where2comm, and V2VAM. The latest download links are available at https://github.com/DerrickXuNu/OpenCOOD.

We host the same dataset and model weights needed for the evaluation of SOMBRA and LUCIA, the latest link can be found at our GitHub repo: https://github.com/WiSeR-Lab/SOMBRA_LUCIA:

You can also use:

```
./scripts/data_model_download.sh
```

to download and extract the data and model weights in to `./assets/`.

The dataset after decompression has the following structure:

```
test
|-- 2021_08_18_19_48_05
|   |---- data_protocol.yaml
|   |---- 1045
|   |---- 1054
|...
|-- scenario_timestamp_id
|   |---- data_protocol.yaml
|   |---- agent_id_1
|   |---- ...
|   |---- agent_id_n
```

Note: *use the data root directory to specify dataset path (i.e., `--data_dir /some_path/test/`). When `--data_dir <path>` is passed to `cp_attack.py`, it overwrites the `validation_dir` specified in the model yaml config.*

The model weights after decompression has the following structure:

```
opv2v_weights
|-- attfusion
|   |---- latest.pth
|   |---- config.yaml
|-- coalign
|   |---- net_epoch15.pth
|   |---- config.yaml
|-- v2vam
|   |---- latest.pth
|   |---- config.yaml
|-- where2comm
|   |---- net_epoch50.pth
|   |---- config.yaml
```

Note: *use the model root directory to specify model path (i.e., `--model_dir /some_path/model_name/`). Also, the model name argument (i.e., `--model <AttentiveFusion/CoAlign/V2VAM/Where2comm>`) is case-sensitive and is used to build the correct attention-hijacking loss for SOMBRA.*

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *SOMBRA achieves high success rates in targeted object removal (TOR) and mass object removal (MOR) attacks against state-of-the-art cooperative perception models. This is demonstrated by experiments (E1) and (E2) in Section 6.2 of the paper.*

(C2): *SOMBRA outperforms prior art in terms of attack success metrics. This is also demonstrated by experiments (E1) and (E2) in Section 6.2 of the paper.*

(C3): *SOMBRA remains effective in the traffic jam scenario with varying number of CAVs. This is demontrated by experiments (E3) in Section 6.3 of the paper.*

(C4): *LUCIA mitigates the impact of SOMBRA by restoring the detection and perception quality measured by mAP. This is demonstrated by experiments (E4) in Section 6.4 of the paper.*

A.4.2 Automated Evaluation Scripts

- *We aggregated dataset and model downloading process, and evaluation commands for different models into scripts located in* `./scripts/`
- *Use* `data_model_download.sh` *to download and process the dataset and weights.*
- *Use* `e1_e2.sh` *to obtain results for C1 and C2. Use* `e3.sh` *and* `e4.sh` *for C3 and C4, respectively.*
- *The results are stored under the same directory as the model weights, in* `./assets/opv2v_weights/<model>`.

A.4.3 Experiments

(E1): Targeted / Mass Object Removal [1 human-hour + 4 compute-hours]:
Preparation: *Download the OPV2V test split and pretrained models. Set the* `-data_dir` *and* `-model_dir` *arguments accordingly.*
Execution: *Run the following command:*

```
python cp_attack.py \
--model_dir <path_to_model> \
--model <cp_model_name> \
--data_dir <path_to_opv2v_test> \
--attack_mode <mor/tor> \
--loss sombra \
(--target_id <random/in/out>)
```

Alternatively, you can use `./scripts/e1_e2.sh` *to obtain the results.*
Results: *Check the saved results in the model directory. Verify the attack success rate matches the reported values in Table 1, 2 of the paper.*

(E2): Comparison with Prior Art [1 human-hour + 4 compute-hours]:
Preparation: *Same as (E1).*
Execution: *Run the following commands for prior art:*

```
python cp_attack.py \
--model_dir <path_to_model> \
--model <cp_model_name> \
--data_dir <path_to_opv2v_test> \
--attack_mode <mor/tor> \
--loss pa \
(--target_id <random/in/out>)
```

Alternatively, you can use `./scripts/e1_e2.sh` *to obtain the results.*
Results: *Compare the attack success rate metrics with SOMBRA, as shown in Table 1, 2 of the paper.*

(E3): Traffic Jam Case Study [2 human-hours + 4 compute-hours]:
Preparation: *Download the traffic jam dataset. Generate perturbations:*

```
python cp_attack.py \
--model_dir <path_to_model> \
--model <cp_model_name> \
--data_dir <path_to_traffic_jam> \
--attack_mode mor \
--loss sombra \
--save_perturb
```

Rename the folder storing perturbed features to `adv_feature`*. Also, verify that the attack success rates match the reported values in Table 3 of the paper.* Alternatively, you can use `./scripts/e3.sh` *to obtain the results.*
Execution: *Run the case study for analyzing adversarial signal propagation:*

```
python case_study.py \
--model_dir <path_to_attfusion> \
--data_dir <path_to_traffic_jam>
```

Results: *Verify the victim's attention paid to the attacker matches the pattern in Figure 9 of the paper, which is found in* `attn` *column of the result csv* `diff_cav.csv`

(E4): Defense Evaluation [2 human-hour + 8 compute-hours]:
Preparation: *Same as (E1).*
Execution: *Run the following command with defenses enabled:*

```
python cp_attack.py \
--model_dir <path_to_model> \
--model <cp_model_name> \
--data_dir <path_to_opv2v_test> \
--attack_mode <mor/tor> \
--loss <sombra/pa> \
(--target_id <random/in/out>) \
(--defense or --robosac)
```

Alternatively, you can use `./scripts/e4.sh` *to obtain the results.*
Results: *Verify the reduced attack success rates with LUCIA and ROBOSAC, as reported in Table 4, 5 of the paper.*

A.5 Notes on Reusability

The artifact is modular and can be extended to evaluate other cooperative perception models or attack strategies. The `cp_attack.py` script can be adapted to test new loss functions or defenses. The traffic jam case study framework can be reused for other scenarios involving cooperative perception.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Await() a Second: Evading Control Flow Integrity by Hijacking C++ Coroutines

Marcos Bajo
CISPA Helmholtz Center for Information Security

Christian Rossow
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This appendix describes the artifacts from our paper on Coroutine Frame-Oriented Programming (CFOP), a novel code reuse attack that exploits vulnerabilities in C++ coroutines. CFOP enables attackers to hijack program execution and manipulate data, even in CFI-protected environments.

Our artifact collection consists of four PoC CFOP exploits on sample programs:

- *gcc*: showcases how to call arbitrary functions with arbitrary arguments using a *silver gadget* in a program compiled with GCC.
- *clang*: showcases how to call arbitrary functions with arbitrary arguments using a *silver gadget* in a program compiled with Clang/LLVM.
- *msvc*: showcases how to call arbitrary functions with arbitrary arguments in Windows, compiled with MSVC.
- *doa*: showcases how to leverage a Data Only Attack (DOA) for altering the program execution without hijacking any frame pointers, reading from an arbitrary file and compiled with GCC.

In addition, we showcase two CFOP exploits in real-world programs:

- *vulnerable_serenityos*: an exploit for SerenityOS after reintroducing the CVE-2021-4327 vulnerability. Showcases Infinite Coroutine Chaining (ICC).
- *vulnerable_scylladb*: an exploit for ScyllaDB with an incorporated vulnerability and a modification in the database client to exploit it. Showcases *golden gadgets* to call arbitrary functions with arbitrary arguments.

Every artifact program has been compiled with Intel CET (and Control Flow Guard (CFG) for *msvc*). Therefore, the exploits work while considering the restrictions introduced by these schemes: 1) return addresses in the stack can never be modified, and 2) the control flow cannot be redirected to any address that is not the start of a function.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifacts do not pose any security, privacy, or ethical risks for evaluators while executing them. The provided exploits are limited to the programs we prepared and do not perform any destructive actions.

A.2.2 How to access

We provide access to every previously described artifact in our in our Zenodo record (*10.5281/zenodo.14738035*) and our *GitHub repository*. The contained README files provide an overview of the artifacts and how to run them.

A.2.3 Hardware dependencies

Our artifacts are prepared to run on standard x86-64 machines and do not require any special hardware features. The exploits are designed to circumvent Intel CET and CFG restrictions and can be tested on any modern Windows/Linux machine (see A.2.4); we do not require evaluators to run them on a machine with CET support.

Optionally running the artifacts with an *active* enforcement of these schemes requires additional hardware support. Specifically, evaluators need a CPU with Intel CET support, that is, Intel Tiger Lake (11th gen) or newer. We recommend a machine with at least 8GB of RAM to compile ScyllaDB and SerenityOS smoothly; we used a machine with 32GB of RAM and a 20-core i9-12900H CPU for our experiments. Around 50 GB of free disk space are needed in total throughout the experiments.

For the purpose of evaluating the artifacts in an environment that meets every hardware and software requirement (including the optional *active* CET enforcement), we provide access to our testing machine via SSH. This machine is restricted for evaluators during the USENIX artifact evaluation period and will be disabled after the evaluation. We incorporate to this artifact submission a WireGuard VPN configuration file to access the machine. The file `AECONNECT.md`

in the repository provides instructions on how to connect to the machine.

A.2.4 Software dependencies

The artifacts have been tested Ubuntu 24.04 and Windows 11 (only for *msvc*).

Every exploit is prepared to run inside a Docker container and can be tested on any Linux distribution with Docker installed; specific software requirements for every artifact are automatically met inside each container. The exception is the Windows *msvc* exploit, that requires a Windows 10 or newer machine with MSVC installed.

A.2.5 Benchmarks

None.

A.3 Set-up

Every artifact in our collection includes its own README file (six in total) with detailed instructions on how to set up their own environment and run the corresponding exploit.

A.3.1 Installation

Docker is required to run the artifacts. For the Windows *msvc* exploit, Python3 is also required. Since each of the six artifacts are independent, here we detail the process of running the *gcc* exploit as an example.

We recommend running the artifacts in our experimental machine. Alternatively, to kickstart this evaluation, clone the repository and install Docker in any Linux machine.

A.3.2 Basic Test

1. Build the docker image:

   ```
   docker build -t cfop_gcc_poc .
   ```

2. Run the docker container

   ```
   docker run -privileged -security-opt
   seccomp=unconfined -it cfop_gcc_poc
   ```

3. Once inside the container, navigate to where the PoC files are saved

   ```
   cd opt/pocs/
   ```

4. Compile the coroutine program

   ```
   make
   ```

5. Run the exploit script

   ```
   python3 gccpocexploit.py
   ```

A successful exploitation results in the program printing *"ARBITRARY CODE EXECUTION!"* on screen, even when this code was not present in the original program.

A.4 Evaluation workflow

A.4.1 Major Claims

Our major claims are:

(C1): CFOP techniques can be used to exploit programs compiled with the latest versions of the three major compilers: GCC, Clang/LLVM, and MSVC.

(C2): Exploits using CFOP work while considering the restrictions introduced by Intel CET and Control Flow Guard CFI schemes: 1) return addresses in the stack can never be modified, and 2) control flow cannot be redirected to any address that is not the start of a function.

A.4.2 Experiments

We recommend following the steps detailed in the README of each artifact to reproduce the results of the paper.

(E1): *[gcc]* [5 human-mins + 1 compute-mins + 4GB disk]:
Preparation: Follow steps 1 to 4 at A.3.2.
Execution: Run the exploit script using `python3 gccpocexploit.py`.
Results: The program should print *"ARBITRARY CODE EXECUTION!"* on screen.

(E2): *[clang]* [5 human-mins + 1 compute-mins + 4GB disk]:
Preparation: Build the Docker image using `docker build -t cfop_clang_poc .`; run the container using `docker run -privileged -security-opt seccomp=unconfined -it cfop_clang_poc`; navigate to the PoC files using `cd opt/pocs/`; and compile the coroutine program using `make`.
Execution: Run the exploit script using `python3 clangpocexploit.py`.
Results: The program should print *"ARBITRARY CODE EXECUTION!"* on screen.

(E3): *[msvc]* [5 human-mins + 1 compute-mins + 1GB disk]:
Preparation: Acquire a Windows 10 or Windows 11 system. Install Python3. If necessary, update the path where the *.exe* file is located in line 10 of the scripts *msvc_silver_exploit.py* and *msvc_call_exploit.py*.
Execution: Run the first exploit using `python3 msvc_call_exploit.py`. Then, run the second exploit using `python3 msvc_silver_exploit.py`.
Results: For the first exploit, the calculator program (*calc.exe*) gets executed. For the second exploit, the parameters values are printed on screen with clearly arbitrary values (e.g., 0x4242424242424242).

(E4): *[doa]* [5 human-mins + 1 compute-mins + 4GB disk]:
Preparation: Build the Docker image using `docker build -t cfop_doa_poc .`; run the container using `docker run -privileged -security-opt seccomp=unconfined -it cfop_doa_poc`; navigate to the PoC files using `cd opt/pocs/`; and compile the coroutine program using `make`.

Execution: Run the exploit script using `python3 fileopening.py`.
Results: The program prints the contents of the file */etc/hosts* on screen, even when this filename was not present in the original program.

(E5): *[vulnerable_serenityos]* [10 human-mins + 30 compute-mins + 22GB disk]:
Preparation: We release the modified source code responsible of building SerenityOS with the CVE-2021-4327 vulnerability. With the goal of simplifying testing, we build and run the Ladybird inside a Ubuntu 24.04 docker - for which we provide the corresponding Dockerfile. If not using our experimental machine, since Ladybird is a program with a GUI, we recommend installing X-Server or a similar system in the host system. Our Docker container takes care of the rest of dependencies.
Execution: Build the Docker image using `sudo docker build -t vulnerable-serenityos .`; run the docker container and attach to it (this command is long and can be found in the README file); once in the container, you can build and run Ladybird `./Meta/serenity.sh run lagom ladybird`;
Once Ladybird is running, the GUI will be shown in screen. At this moment, we can navigate to the URL *file:///cfop/Base/home/anon/exploit.html*. Upon visiting this website, the command line will leak some address. This address is needed to be entered in the browser input as to run the exploit.
Results: As a result of running the exploit, the name of the current user will be printed on screen three times (executing *execve("whoami")*).

(E6): *[vulnerable_scylladb]* [10 human-mins + 40 compute-mins + 34GB disk]:
Preparation: ScyllaDB is made of numerous submodules - hosted in multiple repositories. In addition, its internal build system depends on such submodules for querying additional files at runtime, so it is not possible to offload all the code of ScyllaDB to one single repository. For this reason, our build system consists of 1) A *cfop_setup.sh* script, that clones a certain version of ScyllaDB from its official GitHub repository, patches the code with our vulnerability and exploit, and then builds ScyllaDB; and 2) a folder *cfop_mods* that incorporates the files that patch scyllaDB.
A particular OS version is not needed to run ScyllaDB; every dependency is included in its internal docker setup.
Execution: Run the build script *cfop_setup.sh*. As a result, you will have a new repository *vulnerable_scylladb*; compile ScyllaDB inside the docker container (command can be found in the README, as it is long); attach to the docker container using `podman container attach <id>`; run ScyllaDB from inside the container using `./build/release/scylla -workdir tmp -smp 8 -memory 4G -developer-mode=1`;
Once the ScyllaDB instance is running, we can launch the client program and trigger the exploit: `python3 tools/cqlsh/bin/cqlsh.py`; after it asks for input, introduce `EXPLOITPAYLOAD;`.
Results: Upon sending the previous input, the ScyllaDB instance will stop and show the current user's name on screen (as it runs *execve("bin/sh", "-c /usr/bin/whoami")*).

A.5 Notes on Reusability

The exploits we provide are representative of the attacks describe in the paper and showcase every Coroutine Frame Oriented Programming (CFOP) technique. We intend that these exploits serve as a starting point for replicating the attacks in other programs.

Here we detail which exploits showcase which CFOP technique, serving as a reference for future testers. We recommend consulting the corresponding exploit code to help develop exploits for other coroutine programs:

- *gcc*: Silver gadgets for arbitrary calls with multiple arbitrary arguments in GCC.
- *clang*: Silver gadgets for arbitrary calls with multiple arbitrary arguments in Clang/LLVM.
- *msvc*: Silver gadgets for arbitrary calls with multiple arbitrary arguments in MSVC.
- *doa*: Data Only Attack (DOA) for altering the program execution without hijacking any frame pointers.
- *vulnerable_serenityos*: Infinite Coroutine Chaining (ICC) for multiple arbitrary calls.
- *vulnerable_scylladb*: Golden Gadgets for arbitrary calls with arbitrary arguments.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: System Register Hijacking: Compromising Kernel Integrity By Turning System Registers Against the System

Jennifer Miller*, Manas Ghandat*, Kyle Zeng*, Hongkai Chen*, Abdelouahab (Habs) Benchikh*
Tiffany Bao*, Ruoyu Wang*, Adam Doupé*, Yan Shoshitaishvili*
*Arizona State University
{jmill,mghandat,zengyhkyle,hongkai.chen,abenchik,tbao,fishw,doupe,yans}@asu.edu

A Artifact Appendix

A.1 Abstract

Our paper proposes System Register Hijacking (SRH) as a class of exploitation techniques. As part of the paper, we performed several experiments to determine the prevalence of such gadgets and their applicability. To this end, our artifacts include the following components:

- Scripts for measuring the occurrence of system instructions that may serve as SRH gadgets.
- Kernel module-based PoCs used to demonstrate individual techniques.
- Modified exploits included in our evaluation.
- The setup for reproducing the case study on bypassing FineIBT.
- An implementation of an AVX timing side channel, described in previous work but currently lacking an open-source implementation: https://arxiv.org/abs/2304.07940, usable for breaking KASLR on modern systems.

A.2 Description & Requirements

Our evaluation was primarily conducted on three systems: 1. AMD Ryzen 7 PRO 6850U with 32GB RAM, 2. AMD Ryzen Threadripper 7960X with 32GB RAM, 3. 11th Gen Intel Core i7-1185G7 with 32GB RAM, all of which running Ubuntu 22.04. However, a single Intel x86-64 system with 32GB+ of RAM should be sufficient for evaluation.

A.2.1 Security, privacy, and ethical concerns

Reviewers should take precautions when working with the kernel exploits included as part of our evaluation. Running these exploits outside of a virtualized environment on unpatched Linux kernel versions may result in system instability.

A.2.2 How to access

Our artifacts can be accessed at the following location: https://doi.org/10.5281/zenodo.14728440. The link is associated with a compressed archive of the artifacts. Decompressing and extracting that file will yield an artifacts directory containing a README file that further describes the organizational structure of the artifacts.

A.2.3 Hardware dependencies

Our artifacts expect evaluators to have a modern x86-64 system. The PoCs expect to be ran using hardware-accelerated virtualization features supported by the major x86-64 CPU vendors. Our FineIBT evaluation requires access to a bare-metal x86-64 system with an Intel CPU supporting the IBT feature.

A.2.4 Software dependencies

Our artifacts require that evaluators have access to a Linux system with Docker (https://www.docker.com/) installed. We expect that evaluators to be able to install commonly used software packages on their system, e.g., qemu-system, gcc, and python3.

A.2.5 Benchmarks

None.

A.3 Set-up

Each component should provide either a script that handles software dependencies for the evaluator, or rely only on software packages that are widely used.

A.3.1 Installation

The only major dependency that will not be installed by setup scripts for the experiments themselves is Docker. Evaluators should follow the instructions provided on

Docker's website https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository to set up Docker if they do not already have it installed.

A.3.2 Basic Test

As a basic test, evaluators should attempt to pull and run an Ubuntu 22.04 Docker image. This can be done by running the "docker pull ubuntu:22.04" and "docker run -it ubuntu:22.04 bash" commands. The result of these commands should be that the evaluator has a Bash shell in an Ubuntu 22.04 container.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Ability of "swapgs Stack Pivoting" to Bypass FineIBT. This is proven by Experiment (E4), which relates to Section 8.3 of the paper.

A.4.2 Experiments

(E1): [Gadget Analysis] [10 human-minutes + 8 compute-hours + 3GB disk]: Downloads kernel packages from Ubuntu and Fedora repositories, analyzes the kernel images for System Register Hijacking gadgets, and produces a csv file of resulting gadgets
How to: In the "analyzer" directory of the artifacts, execute the "analyze.sh" script. It will run the entire analysis.
Results: The README describes how to use the scripts that were used to generate the tables found in the paper, the tables in the paper should closely match the output from this analysis.

(E2): [Kernel Module PoCs] [20 human-minutes + 1 compute-hour]:
How to: In the "eval-module" directory there is a subdirectory for each architecture: "x86" and "arm". Each architecture has its own setup. The x86 directory contains a "setup.sh" script that will install several apt and python packages, fetch an Ubuntu kernel image, and build a debian filesystem image. From there the "run-50.sh" script can be used to semi-automate the evaluation of individual technique reliability. For ARM, the "setup" directory in the directory of each technique contains a "startvm" script that will boot a vm containing the PoC, which can be run by executing "./poc" in the VM's shell.
Results: Results from running the experiment can be compared with the results included in the paper in Table 2.

(E3): [CVE-Based PoCs] [10 human-minutes + 4 compute-hours]:
How to: In the "eval-cve" directory there is a script called "run-eval.sh" that will run the exploit for each kCTF based CVE we modified 50 times and collect the success rates. The script can be run with either "modified" or "original" as an argument to evalutate the stability of either our modified exploits or the exploits using the control flow hijacking approach found in the original exploits. The older CVE, which was pulled from the RetSpill (https://github.com/sefcom/RetSpill) dataset, can be found in the "eval-cve-retspill" directory. This directory contains a script, "start.sh" that will spawn a containerized environment to evaluate the exploit in.
Results: The results of this experiment can be compared with the results found in Table 3 of the paper.

(E4): [FineIBT Evaluation] [2 human-hours + 1 compute-hour]:
How to: In the "eval-fineibt/vuln-experiment" directory, there is an "iso" directory that should be turned into an ISO file via the "grub-mkrescue" command found in the README. The ISO should be loaded onto a bootable drive and be booted into on a baremetal system. Baremetal is necessary because IBT virtualization is not currently available. Once booted into the ISO, the evaluator will be placed into an initramfs containing our modified exploit at "/exploit".
Results: The exploit should sucessfully and reliably print the flag and then attempt to shut down the system.

(E5): [Sidechannel] [10 human-minutes + 5 compute-minutes]: ...
How to: In the "sidechannel" directory, run "start.sh". This will build and drop the evaluator into a containerized environment. In the container the "run.sh" script can be used to boot a VM in which to test the side-channel. Running "./kaslr" in the VM should result in the KASLR base being printed, which can be compared with the KASLR base that gets logged to the console on boot.
Results: The side-channel should leak the base address of the kernel.

A.5 Notes on Reusability

The header-files used for several of the CVE-based exploits for the 'swapgs Stack Pivoting' technique can be reused to apply the technique to other exploits. The sidechannel implementation provided can be reused to demonstrate the ability to break KASLR in future research.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks

Lukas Maar
Graz University of Technology

Lukas Giner
Graz University of Technology

Daniel Gruss
Graz University of Technology

Stefan Mangard
Graz University of Technology

A Artifact Appendix

A.1 Abstract

This paper shows how side-channel leakage in kernel defenses can be exploited to leak the locations of security-critical kernel objects, enabling reliable and stable attacks on the Linux kernel. By systematically analyzing 127 defenses, we show that enabling any of three specific defenses – strict memory permissions, kernel heap virtualization, or stack virtualization – exposes fine-grained TLB contention patterns. These patterns are then combined with kernel allocator massaging to perform location disclosure attacks, revealing the locations of kernel heap objects, page tables, and stacks.

The artifacts demonstrate the timing side-channel attack and the exploit techniques. For both, we provide a kernel module and programs to perform the experiments.

1. For the timing side channel, we leak the location of kernel heap objects (i.e., `pipe_buffer`, `msg_msg`, `cred`, `file`, and `seq_file`), page tables (all levels), and the kernel stack. While our side channel works on all Intel generations between the 8th and 14th, we recommend evaluating on Intel 13th generation, as we have mainly evaluated on this one. While our side channel works on Linux kernels between v5.15 and v6.8, we recommend evaluating on the Ubuntu generic kernel v6.8.
2. For the exploit techniques, we perform privilege escalation using the 3 techniques. Inherent to kernel exploitation, we tailor these techniques to the specific Ubuntu generic kernel v6.8.0-38, the required version to evaluate these techniques.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

For evaluating the timing side channel, the experiments can be used in kernel exploitation of memory-corruption attacks, as they allow the location of kernel objects be leaked on Intel CPUs. This raises potential ethical concerns.

For evaluating the exploit techniques, the artifacts might result in destructive steps. While we introduce an exploit primitive via a kernel module and do not provide methods to compromise systems in the wild, the experiment using this primitive may cause system crashes. However, during our evaluation, we have not encountered a single system crash.

A.2.2 How to Access

We provide the source code (github and zenodo) for performing the timing side channel.

A.2.3 Hardware Dependencies

A Linux system running on an Intel CPU between 8th and 14th generation. However, we recommend running the evaluation on a 13th generation Intel, as this is what we have mainly evaluated on. We have observed a trend that newer Intel CPUs and better cooling tend to give more stable results.

A.2.4 Software Dependencies

While our disclosure attacks should generically work on Linux kernels, our experiments are tailored to Ubuntu Linux kernels between v5.15 to v6.8. As reference, we have mainly evaluated on the generic Ubuntu kernel v6.8.0-38 (and the kernel with `CONFIG_SLAB_VIRTUAL` of v6.6[1]).

One part of the artifact evaluation is to insert a kernel module that requires *root privileges*. This module is required to obtain the ground truth of the kernel object's location as well as for providing the exploit primitive for the exploit techniques. We tested our kernel module on Ubuntu Linux downstream kernels v5.15, v6.5, and v6.8 and kernel v6.6. For other kernels that have different config files (or other downstream changes) our implemented module may not do what we intended. Specifically, in order to obtain the location of kernel objects, we redefined and reimplemented structures

[1] https://lore.kernel.org/all/CAHKB1wLetbLZjhg1UVhA1QwZHo226BRL=Khm962JEfh0F+CVbQ@mail.gmail.com/T/

and functions. We did this because some functions used to access kernel data structures are implemented as inline functions, e.g., `ipc_obtain_object_check`, which prevented us from calling these functions directly. Another reason is that some structs, e.g., `msg_queue`, are defined in c files, which also prevents us from using these struct definitions.

While this artifact evaluation include experiments exploiting leakage for all defenses, we only recommend to reproduce experiments exploiting leakage from **D1** and **D3**. This is because reproducing leakage from **D2** requires a Linux kernel compiled with `CONFIG_SLAB_VIRTUAL`, i.e., the intended v6.6 used by Google's KernelCTF. We encountered driver crashes during boot due to incompatibilities, requiring additional engineering effort. However, all of our heap location leakage attacks should work directly by exploiting **D2** when swapping the base address from the DPM to the virtual heap.

Due to the nature of kernel exploitation in general, our exploit techniques depend on the kernel version. Therefore, we only provide the end-to-end attack for the exact Ubuntu Linux kernel v6.8.0-38 (and the kernel with `CONFIG_SLAB_VIRTUAL` of v6.6). The exact version is needed, e.g., for the control-flow hijacking attack, as the ROP chain varies between versions. Similarly, the other two exploit techniques require internal version-dependent kernel information. Curically, all information can be obtained as an unprivileged user but requires engineering effort.

A.3 Set-Up

A.3.1 Installation

The installation works as following:

1. Clone our github repository (github) to `/repo/path` directory.
2. Change directory to `/repo/path`.
3. Select in `./lkm.c` either `V5_15`, `V6_5`, `V6_6`, or `V6_8`, depending on your running Ubuntu Linux kernel version.
4. Execute `make init` to build the kernel module and all experiments and insert the kernel module.

A.3.2 Basic Test

Before starting the experiments, determine the TLB hit thresholds on your CPU as follows. It is important to ensure that the background noise is as low as possible before starting this basic experiment, as described in A.5.

1. Change directory to `/repo/path/generic`.
2. Execute `./threshold_detection.elf` prints `[+] detected thresholds: <THRESHOLD> <THRESHOLD2>`, where `THRESHOLD` is the threshold for capturing 97 % of all hit timings of mapped addresses and `THRESHOLD2` is the threshold for the minimum timing of unmapped addresses.
3. Repeat this a few times and write the most consistent result to `THRESHOLD` and `THRESHOLD2` in `/repo/path/include/tlb_flush.h` and recompile with `make build` in `/repo/path`.

A.4 Evaluation Workflow

A.4.1 Major Claims

We provide artifacts verifying the following claims:

(C1): We demonstrate to leak the base address of the DPM, i.e., `page_offset_base`. This is proven by **(E1)** described in Section 6.2 **Leaking Coarse-Grained Kernel Section**.

(C2): We demonstrate to leak the base address of the memory location used by `vmalloc`. This is proven by **(E2)** described in Section 6.2 **Leaking Coarse-Grained Kernel Section**.

(C3): We demonstrate to leak the base address of the virtual memory mapping `vmemmap`. This is proven by **(E3)** described in Section 6.2 **Leaking Coarse-Grained Kernel Section**.

(C4): We demonstrate that exploiting **D1** allows to leak the page-aligned location of `msg_msg`. This is proven by **(E4)** described in Section 5.1 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C5): We demonstrate that exploiting **D1** allows to leak the page-aligned location of `file`. This is proven by **(E5)** described in Section 5.1 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C6): We demonstrate that exploiting **D1** allows to leak the page-aligned location of `seq_file`. This is proven by **(E6)** described in Section 5.1 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C7): We demonstrate that exploiting **D1** allows to leak the page-aligned location of `pipe_buffer`. This is proven by **(E7)** described in Section 5.1 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C8): We demonstrate that exploiting **D1** allows to leak the locations of page-tables, i.e., Page Table (PT), Page Middle Directory (PMD), and Page Upper Directory (PUD). This is proven by **(E8)** described in Appendix B and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C9): We demonstrate that exploiting **D3** allows to leak the location of the kernel stack. This is proven by **(E9)** described in Section 5.3 and Section 6.2 **Leaking Fine-Grained Locations** and shown in Table 1.

(C10): We demonstrate high reliability of our location disclosure attacks. This is proven by **(E10)** shown in Table 1.

(C11): We demonstrate the **Unlink Primitive** exploit technique. This is proven by **(E11)** described in Section 7.1.

(C12): We demonstrate the **Use-After-Free & Out-Of-Bounds Write** exploit technique. This is proven by **(E12)** described in Section 7.1.

(C13): We demonstrate the **Constrained Write Primitive**

exploit technique. This is proven by **(E13)** described in Section 7.1.

(C14): We demonstrate high reliability of our exploit techniques. This is proven by **(E14)**.

A.4.2 Experiments

Before running the experiments, please perform the set-up in A.3 and read the note in A.5.

(E1): Basic DPM location leakage [10 human-seconds + 1 computer-second]:
How to: Execute `./generic/dpm_leak.elf`.
Results: This experiment outputs the base address of the DPM.

(E2): Basic `vmalloc` memory location leakage [10 human-seconds + 1 computer-second]:
How to: Execute `./generic/vmalloc_leak.elf`.
Results: This experiment outputs the base address of the virtual memory section used for `vmalloc`.

(E3): Basic `vmemmap` memory location leakage [10 human-seconds + 1 computer-second]:
How to: Execute `./generic/vmemmap_leak.elf`.
Results: This experiment outputs the base address of the virtual memory mapping `vmemmap`.

(E4): `msg_msg` location leakage [10 human-seconds + 10 computer-seconds]:
How to: Execute `./heap/msg_msg_leak.elf`.
Results: This experiment outputs the page-aligned `msg_msg` object location.

(E5): `file` location leakage [10 human-seconds + 10 computer-seconds]:
How to: Execute `./heap/file_leak.elf`.
Results: This experiment outputs the page-aligned `file` object location.

(E6): `seq_file` location leakage [10 human-seconds + 10 computer-seconds]:
How to: Execute `./heap/seq_file_leak.elf`.
Results: This experiment outputs the page-aligned `seq_file` object location.

(E7): `pipe_buffer` location leakage [10 human-seconds + 10 computer-seconds]:
How to: Execute `./heap/pipe_buffer_leak.elf`.
Results: This experiment outputs the page-aligned `pipe_buffer` object location.

(E8): Page-table location leakage [10 human-seconds + 20 computer-seconds]:
How to: Execute `./page-table/pt_leak.elf`, `./page-table/pmd_leak.elf`, or `./page-table/pud_leak.elf`.
Results: This experiment outputs the respective location of PT, PMD, and PUD.

(E9): Kernel stack location leakage [10 human-seconds + 2 computer-seconds]:
How to: Execute `./stack/stack_leak.elf`.
Results: This experiment outputs the current kernel stack location.

(E10): Reliable location disclosure attacks [5 human-minute + 2 computer-hours]:
How to: Execute `./eval.sh` (in `heap`, `page-table` and `stack`) and then `./print.py`.
Description: The `./eval.sh` scripts performs between 20 to 100 execution of (E4-9) depending on the experiment, while `./print.py` prints a table which should closely resemble Table 1. As described in A.5, background activity should be minimized in this evaluation.
Results: This experiment outputs the content of Table 1 for **D1** and **D3**.

(E11): Unlink primitive [10 human-seconds + 1 computer-seconds]:
How to: Execute `./attacks/pipe_unlink.elf`.
Limitation: Works with Ubuntu kernel v6.8.0-38. Other versions will most likely lead to a program crash.
Results: Privilege escalation.

(E12): Use-After-Free & Out-Of-Bounds write [10 human-seconds + 1 computer-seconds]:
How to: Execute `./attacks/dirty_page.elf` or `./attacks/advanced_slubstick.elf`.
Limitation: Works with Ubuntu kernel v6.8.0-38 or the v6.6 intended to be used with `CONFIG_SLAB_VIRTUAL`. Other versions will most likely lead to a program crash.
Results: Privilege escalation.

(E13): Constrained write [10 human-seconds + 3 computer-seconds]:
How to: Execute `./attacks/stack_attack.elf`.
Limitation: Works with Ubuntu kernel v6.8.0-38. Other versions will most likely lead to a program crash.
Results: Privilege escalation.

(E14): Reliable exploit techniques [5 human-minutes + 10 computer-minutes]:
How to: Execute `./eval.sh` and then `./print.py`, both in `attacks`.
Results: This experiment shows the success rate of the 3 exploit techniques.

A.5 Notes on Reusability

As described in Section 6.2 **Stress**, the most dominant noise source is CPU frequency fluctuation. Hence, perform all experiments with as little background activity as possible to reproduce the paper's results. We even suggest to perform the experiments on an idle system with no other activity.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Approximation Enforced Execution of Untrusted Linux Kernel Extensions

Hao Sun
ETH Zurich

Zhendong Su
ETH Zurich

A Artifact Appendix

A.1 Abstract

This artifact accompanies the paper on Approximation-Enforced Execution (AEE), a novel concept for ensuring the safe execution of untrusted kernel extensions even in the potential presence of the verifier's soundness bugs. The artifact consists of Linux kernel patches for version 6.7, which implement the offset, size, and tag enforcement, alongside a suite of benchmarks, build scripts, and analysis tools. This appendix provides a detailed roadmap for building the customized kernels, executing functional and performance evaluations, and reproducing the paper's results, including AEE's impact on runtime overhead, binary size, verification time, and the trusted code base (TCB) reduction.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact operates exclusively within QEMU ARM64 virtual machines (VMs) running modified Linux kernels. All evaluations involving untrusted eBPF programs are conducted in isolated VMs, ensuring that no interactions with the host environment occur. Evaluators must refrain from executing modified kernels or any associated exploits outside the QEMU environment. The artifact does not collect or transmit any data, nor does it require or expose sensitive information.

A.2.2 How to access

The complete artifact package—including kernel patches, benchmark workloads, build and execution scripts, pre-built kernel binaries, and disk images—is archived at: `https://doi.org/10.5281/zenodo.15609051`. Evaluators are advised to begin from the root directory of the unpacked archive, which contains all resources necessary for reproduction and evaluation.

A.2.3 Hardware dependencies

The artifact has been tested under the following hardware conditions:

- **CPU:** Host system capable of running QEMU with ARM64 emulation and Pointer Authentication Code (PAC) support (validated on Apple M1).
- **Memory:** At least 16 GB of RAM is recommended to ensure reliable kernel compilation and VM execution.
- **Storage:** A minimum of 50 GB of available disk space is required to accommodate kernel sources, build artifacts, disk images, and experimental results.

A.2.4 Software dependencies

The following software dependencies are required for full functionality and building relevant components:

- **Operating System:** A Linux host system is required to build the customized kernel (validated on Ubuntu 22.04). MacOS systems with Apple Silicon (M1+) are tested for running all the benchmarks in the virtual machine; other systems running on ARM64 are also feasible.
- **Emulator:** QEMU version 7.0.0 or later with ARM64 support (*e.g.*, `qemu-system-aarch64`); tested on QEMU version 9.2.2.
- **Build Tools:** `make`, `flex`, `bison`, `libssl-dev`, `libelf-dev`, `libncurses-dev`, `dwarves` for building the Linux kernel from source.
- **Cross Compiler:** `gcc-aarch64-linux-gnu`; tested with version 11.4.
- **Analysis Tools:** `Python3` and `Rust` for parsing and executing the evaluation scripts.

If the evaluator opts to use the pre-built kernels included in the archive, kernel compilation is not required, and the Linux host prerequisite may be skipped. On Ubuntu/Debian-based systems, the required dependencies can be installed with the following command:

```
apt install build-essential gcc-aarch64-linux-gnu flex
    bison libssl-dev libelf-dev libncurses-dev dwarves
    python3 rustc cargo qemu-system-arm
```

A.2.5 Benchmarks

The artifact includes four benchmark suites that collectively evaluate AEE's effects on program size, verification time, and execution time across various enforcement configurations. All benchmarks are pre-installed in the provided disk image: (1) `linux-progs`: eBPF programs extracted from Linux kernel selftests and samples; (2) `linux-bench`: benchmark programs crafted to stress specific eBPF instruction patterns and corner cases; (3) `katran`: real-world eBPF workloads from the production-level load balancer; and (4)`filter`: CPU-intensive eBPF packet filters used for performance stress testing. These benchmarks are used to quantify the potential overhead introduced by AEE's enforcement.

A.3 Set-up

A.3.1 Installation

All setup steps assume that the evaluator is operating from the root directory of the unpacked artifact archive. The artifact provides automated scripts to minimize manual intervention.

Step 1: Download Linux v6.7 Source.

```
mkdir -p build kernels
wget -P build <download_link>/linux-6.7.tar.gz
```

Step 2: Build Kernel Variants. The following script performs various tasks: (1) it unpacks the Linux 6.7 source tree; (2) it applies AEE kernel patches; and (3) it compiles five kernel variants. The kernel variants include: (1) `clean`: unmodified baseline kernel (no AEE); (2) `all`: kernel with all three AEE mechanisms enabled; (3) `offset`: kernel with only the offset enforcement; (4) `tag`: kernel with only the tag enforcement; and (5) `size`: kernel with only the size enforcement. Note that kernel compilation must be performed on a Linux host. macOS is not supported for this step due to toolchain and compatibility limitations.

```
./scripts/build_kernels.sh full
```

Step 3: Synchronize Kernel Images. After successful compilation, synchronize the kernel images to the designated directory. All resulting images will be available in `./kernels/`:

```
./scripts/sync_kernels.sh
```

Step 4: Disk Image Preparation. A pre-built ARM64 Debian Bookworm disk image is provided in `imgs/`. It includes: (1) pre-installed SSH keys for headless login; and (2) benchmark scripts and execution tools. No additional configuration is required unless re-generating the image from scratch.

A.3.2 Basic Test

To boot the baseline kernel in QEMU:

```
./scripts/boot.sh clean
```

This launches QEMU with the `clean` kernel. You should observe Linux boot messages followed by a login prompt. To login via SSH:

```
ssh -i imgs/bookworm.id_rsa -p 10023 root@localhost
```

To exit the VM safely, execute:

```
shutdown now
```

You may also test other kernel variants:

```
./scripts/boot.sh all
./scripts/boot.sh offset
./scripts/boot.sh tag
./scripts/boot.sh size
```

A.4 Evaluation workflow

A.4.1 Major Claims

The artifact supports the following key claims made in the paper:

(C1): AEE reduces the verifier's trusted code base (TCB) by 4.5x (see Table 1).

(C2): AEE prevents kernel exploitation via verifier soundness bugs (see §6.1 and Table 2).

(C3): AEE incurs 1.2% runtime and 4.8% binary size overhead on average (see Tables 3–5).

A.4.2 Experiments

Each experiment below corresponds to one or more paper claims. All experiments are self-contained and can be run using the provided scripts and disk image.

(E1) TCB Reduction (15 min manual effort). This experiment measures the reduction in the trusted code base achieved by enforcing the approximation produced by the verifier with AEE. The file `scripts/TCB_stats.pdf` reports the lines of code (LOC) for: (1) `V.a`: approximation logic in the verifier; (2) `V.s`: safety checks retained in the TCB; and (3) AEE implementation: runtime enforcement components. The evaluator can (1) open `scripts/TCB_stats.pdf`, and (2) review LOC breakdown for `V.a`, `V.s`, and AEE, and (3) optionally cross-reference the kernel source in `build/linux-6.7/`.

Interpretation: The verifier's approximation logic dominates its complexity. By shifting approximation out of the TCB and into smaller enforcements, AEE reduces the verifier's TCB by 4.5x, supporting Claim (C1).

(E2) Exploit Mitigation (30 min manual + 0.5 hr compute). This experiment evaluates AEE's ability to prevent exploits that bypass verifier checks. Exploit PoCs located in `exploits/` target known bugs in the verifier's approximation logic. To reproduce C2, the evaluator should (1) boot the baseline kernel using `scripts/boot.sh clean`, (2) SSH into the

VM and execute each exploit, (3) repeat this procedure on the AEE-enabled kernel. Please refer to `exploits/README.md` for more specific instructions.

Interpretation: On the unprotected kernel, the exploits compromise kernel integrity. Under AEE, violating accesses are caught at runtime, demonstrating effective mitigation. This validates Claim (C2), as shown in Table 2 and §6.1.

E3.1 Binary Size Overhead (1-2 hr compute). The goal is to quantify the increase in binary size of eBPF programs caused by AEE's runtime enforcement. The script `run_bench.sh bench_size` triggers execution of benchmarks inside QEMU for each kernel variant, measuring the bytecode size of eBPF programs before and after AEE transformations. It uses the disk image, runs the target workloads, and collects raw statistics. The `analyze.sh size` script parses the results and produces the analysis results.

```
./scripts/run_bench.sh bench_size
./scripts/analyze.sh size
```

Interpretation: Results show per-benchmark and average binary size increases. The "impacted average" reflects only modified programs, while the "overall average" includes all inputs. AEE causes a modest increase (~4.8%) consistent with the results shown in Table 3.

E3.2 Execution Time (1–2 hr compute). This part assesses AEE's runtime overhead during the execution of eBPF programs. The script `run_bench.sh bench_exec_time` launches the QEMU VM for each kernel variant, runs each benchmark suite, and saves execution statistics. The `analyze.sh exec_time` script computes per-suite and global averages.

```
./scripts/run_bench.sh bench_exec_time
./scripts/analyze.sh exec_time
```

Interpretation: The output includes execution times per benchmark and enforcement variant. The `all` configuration yields ~1.2% average runtime overhead, with individual enforcements contributing differently. These results support the statistics shown in Table 4.

E3.3 Verification Time (1–2 hr compute). Finally, we measure the increase in verification time incurred by enabling AEE. The `run_bench.sh bench_verify_time` script executes each benchmark and records the time spent on verification, including AEE transformations. The `analyze.sh verify_time` script computes total and average timing per kernel configuration.

```
./scripts/run_bench.sh bench_verify_time
./scripts/analyze.sh verify_time
```

Interpretation: The results show that while AEE increases verification time due to transformation overhead, the increase remains acceptable for practical deployment. Table 5 in the paper summarizes these findings.

Note: For additional guidance and expected outputs, please refer to `README.md` in the artifact archive.

A.5 Notes on Reusability

AEE is implemented as a modular set of patches applicable to Linux v6.7 and can be adapted to future versions with certain integration effort. Each enforcement mechanism—tag, offset, and size—is isolated, allowing evaluators to enable or disable them independently. The infrastructure is compatible with privileged execution contexts and exposes hooks (e.g., `BPF_F_TEST_REWRITES`) for custom instrumentation and introspection. Researchers may repurpose the artifact to develop or test alternative verifier-hardening techniques or runtime policies for eBPF safety enforcement.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: EKC: A Portable and Extensible Kernel Compartment for De-Privileging Commodity OS

Jiaqin Yan[*†], Qiujiang Chen[†], Shuai Zhou[†], Yuke Peng[†], Guoxing Chen[*], Yinqian Zhang[†]
[*]Shanghai Jiao Tong University, [†]Southern University of Science and Technology
yan2364728692@gmail.com, {12012211, zhous2021, pengyk}@mail.sustech.edu.cn,
guoxingchen@sjtu.edu.cn, yinqianz@acm.org

A Artifact Appendix

A.1 Abstract

The artifact includes the following components:

- `RustEKC_src.zip`: Contains the complete source code and documentation for RustEKC, consistent with the open-source repository. It enables users to examine the EKC's internal design, modify the implementation, and recompile it for integration into custom applications.
- `payload_src.zip`: Provides the source code of the supported OS kernels—rCore, FreeRTOS, and TinyLinux—demonstrating how each integrates with EKC. Users can freely inspect, modify, and rebuild these kernels as needed.
- `RustEKC_artifact.zip`: Offers pre-compiled binaries for execution on the QEMU platform, allowing users to evaluate RustEKC without setting up a full build environment. All binaries are compiled from the sources in `RustEKC_src.zip` and `payload_src.zip`, and are intended for basic testing.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The execution of this artifact does not pose any risks to the evaluator's system security, data privacy, or ethical considerations. Once the environment is set up and installed, the artifact can be executed and tested without requiring Internet access.

A.2.2 How to access

This submission complies with the Open Science guidelines of USENIX Security '25. All relevant artifacts have been publicly released on `Zenodo`:

doi.org/10.5281/zenodo.15534623

Additionally, the latest working copy of the artifact for is maintained in an open-source repository. For artifact evaluation, a stable release is available at:

github.com/EmbeddedKC/RustEKC/releases/tag/AEC

Future updates and bug fixes will be published through this repository.

A.2.3 Hardware dependencies

The artifact supports both simulation and hardware-based testing.

Simulation: An x86 host machine is sufficient.

Hardware (board) testing: An x86 host is required, along with one or more of the following development boards:

- Raspberry Pi 4B (for the AArch64 instruction set)
- Allwinner Nezha D1-H (for the RISC-V instruction set)

In addition, UART and USB cables are needed for FEL flashing and serial debugging.

A.2.4 Software dependencies

The following software tools are required:

- `QEMU` 6.2.0 — for emulated testing.
- `xfel` — for deploying EKC on the Allwinner D1 board.
- Pi4 flashing utility — for Raspberry Pi 4B support.
- `PuTTY` — for serial port debugging.

The following toolchains are required:

- `Rust` nightly-2023-06-25
- `gcc-arm-none-linux-gnueabi` 2021-07
- `gcc-riscv64gc-unknown-none-elf` 10.2.1
- `gcc-aarch64-none-elf` 4.3.2

A.2.5 Benchmarks

All test examples and benchmarks are included in the artifact.

A.3 Set-up

A.3.1 Installation

To run the artifact in an emulator, QEMU must be installed. The following shell script provides a quick installation method on Ubuntu 22.04:

```
sudo apt update
sudo apt install -y build-essential \
  make git pkg-config libglib2.0-dev \
  zlib1g-dev libpixman-1-dev \
  ninja-build python3 python3-pip \
  libncurses5-dev libspice-server-dev \
  libncursesw5-dev libspice-protocol-dev \
  libcap-dev libvirglrenderer-dev \
  libsdl2-dev libgtk-3-dev libattr1-dev

git clone https://gitlab.com/qemu-project/
    qemu.git
cd qemu
git checkout v6.2.0

./configure --target-list=arm-softmmu,\
aarch64-softmmu,riscv64-softmmu

make -j$(nproc)
sudo make install
```

To compile the artifact, you must install Rust and the required cross-compilation toolchains. Follow these steps:

Install `Rust nightly-2023-06-25`[1].

Install `gcc-arm-none-linux-gnueabi` 4.3.2[2].

Install `riscv64gc-unknown-none-elf-gcc` 10.2.0[3].

Install `aarch64-none-elf-gcc` 2021-07[4].

Install `xfel`[5] to run EKC on the Allwinner D1H board.

A custom SD card flashing tool for Raspberry Pi 4B is included in the artifact package.

Finally, download and extract `RustEKC_src.zip` and `RustEKC_artifact.zip` from Zenodo.

A.3.2 Basic Test

Open the `RustEKC_artifact` directory to access a set of pre-compiled binary files that can be executed directly using QEMU. To run a demo, navigate to the corresponding subdirectory and run one of the following commands:

```
make INFO=1  % Verbose output
# or
make INFO=0  % Minimal output
```

This will produce QEMU output, either with full logging (`INFO=1`) or minimal output (`INFO=0`). Refer to the expected output described in the `Explanation.md` file within each demo folder. This basic test only requires QEMU to be installed on the host machine.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *EKC can be deployed across multiple OS kernels and instruction set architectures (ISAs). This is proven by the experiment (E1) and (E2) described in section 3.1 and Table 4.*

(C2): *EKC can protect user data across various application scenarios. This is proven by the experiment (E1) and (E2) described in section 3.4.*

(C3): *EKC has been successfully deployed on multiple development boards. This is proven by the experiment (E1) and (E2) described in section 3.1 and Table 4.*

A.4.2 Experiments

(E1): *[Compile EKC for different platforms] [1 human-hour]:*

How to: Modify the configuration in the source code and compile the binaries for execution in QEMU.

Preparation: Open `RustEKC_src` and edit the `Makefile`. Adjust the following parameters: `BOARD` specifies the target platform, selectable from `codes/arch`. The `TARGET` variable specifies the Rust compilation target. Typical values include:

- arm32-none-linux-gnueabi
- aarch64-unknown-none
- riscv64gc-unknown-none-elf

Execution: For each configuration, run `make env` followed by `make all` to compile EKC for the selected platform. Use `make run` to launch EKC under QEMU.

Results: EKC will initialize on the selected platform and terminate with a panic, as expected, due to the absence of a payload.

(E2): *[Run EKC with different payloads in QEMU] [1 human-hour]:*

Preparation: Compile each payload following the instructions in its source directory. In the `Makefile`, assign the compiled binary to the `PAYLOAD` variable. Configure `BOARD` and `TARGET` accordingly.

Execution: After updating the configuration, run `make env` and `make all`, then use `make run` to launch EKC with the selected payload on QEMU.

[1] https://www.rust-lang.org/tools/install
[2] https://ftp.gnu.org/gnu/gcc/gcc-4.3.2/
[3] https://github.com/riscv-collab/riscv-gnu-toolchain
[4] https://developer.arm.com/downloads/-/gnu-a/10-3-2021-07
[5] https://github.com/xboot/xfel

Results: EKC will execute with the specified payloads. The output should match the results observed in `RustEKC_artifact`.

(E3): *[Run EKC with payloads on development boards] [2 human-hours]:*
Preparation: Take Raspberry Pi 4B as an example. Recompile both EKC and the payload for the AArch64 Raspberry Pi platform to produce binaries. Use the provided flash tool at `tools/pi-fel-tool` to write the binaries to an SD card, then insert it into the board.
Execution: Connect the board to your computer via a serial cable and open a terminal using PuTTY. Power on the board to observe its output in the PuTTY console.
Results: The output should be consistent with the QEMU results observed in experiment (E2).

A.5 Notes on Reusability

You may deploy your own payload with EKC on a specific platform. To integrate EKC into a custom operating system kernel, follow these steps:

1. **Modify the linker script** (typically `linker.ld`) to set the entry point to the EKC jump address, which is defined in `config.rs`.

2. **Include the EKC API library**, available from the GitHub:

   - For Rust projects, add the `mmi` dependency to `Cargo.toml`.
   - For C/C++ projects, add `libekc/include` to the compiler's header search path, and link against the static library `libekc/build/libmmk_arch.a`.

3. **Invoke EKC API functions** during kernel initialization (usually in `start.S`) to configure memory permissions for the `.text`, `.data`, and `.bss` segments.

   By default, EKC grants full access only to the first memory page. Therefore, the memory permissions for these segments must be configured within the first page during initialization.

4. **Replace existing memory management module (if any).** If your OS contains a memory management system, decouple and remove it. All original function calls should be replaced with EKC API equivalents. This may involve non-trivial code refactoring.

   - If your OS lacks a memory management module (e.g., RTOS), this step can be skipped.
   - If your OS uses a decoupled architecture (e.g., microkernel), integration should be relatively straightforward.

5. **Integrate EKC with the trap handler.** If your OS defines a trap or interrupt handler, modify the instructions that set the interrupt vector register (e.g., `stvec` on RISC-V or `VBAR` on ARM) to route through the EKC API.

6. **Compile your kernel to obtain the binary image.** If compilation succeeds, the output binary can be placed at a designated location such as `payload/your_own_OS.img`.

7. **Run your kernel with EKC.** Update the `Makefile` by setting the `PAYLOAD` to the path of your kernel image. You can then launch the system with EKC and utilize its security features via the provided API.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: The Cost of Performance: Breaking ThreadX with Kernel Object Masquerading Attacks

Xinhui Shao[†], Zhen Ling[†,*], Yue Zhang[‡], Huaiyu Yan[†], Yumeng Wei[†], Lan Luo[¶], Zixia Liu[¶], Junzhou Luo[†], Xinwen Fu[§]
[†] Southeast University, Email: {xinhuishao, zhenling, huaiyu_yan, yumeng5, jluo}@seu.edu.cn
[‡] Drexel University, Email: zyueinfosec@gmail.com
[¶] Anhui University of Technology, Email: {lluo, zxliu}@ahut.edu.cn
[§] University of Massachusetts Lowell, Email: xinwen_fu@uml.edu

A Artifact Appendix

A.1 Abstract

In this artifact, we provide a repository of the source code for our symbolic execution engine and the Proof of Concept (PoC) experiment, along with the necessary materials to replicate the experiments described in the paper. To facilitate the evaluation process, we also offer corresponding Docker images, allowing users to set up the experiment environment and perform the related experiments with ease. This document outlines the steps to reproduce the results presented in Sections §6.2 and §A.2 of the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All experiments are conducted within a Docker container, except for one PoC experiment, which is performed on an actual board. The reproducer will need to execute several commands on the host machine during the experiment. These commands are commonly used and safe, ensuring they will not harm the system.

A.2.2 How to access

The code and experimental materials are available on Zenodo (DOI: `10.5281/zenodo.14643155`) and GitHub (`https://github.com/x-codingman/KOM-experiments.git`). Reproducers can either build the Docker images themselves, pull them directly from Docker Hub using the link provided in our GitHub repository, or download them from the Zenodo platform.

*Corresponding author: Prof. Zhen Ling of Southeast University, China.

A.2.3 Hardware dependencies

- **Processor**: We recommend using a machine with two Intel Xeon E5-2620 v2 CPUs (12 cores, 24 threads) to reproduce the experiment. However, comparable hardware may also suffice.
- **Memory**: At least 64GB of RAM.
- **Storage**: At least 256GB.
- **Board (Optional)**: NUCLEO-U575ZI-Q is used as one of the platform in the PoC experiment.

A.2.4 Software dependencies

Most experiments depend on Ubuntu 24.04, git, Qemu, and Docker. For the PoC involving the actual board, STM32CubeIDE is required to reproduce **E3**.

A.2.5 Benchmarks

Target system calls. We get the ThreadX source code (version 6.2.1) from GitHub. To facilitate the system call compilation, we modify the system call wrappers to generate the intermediate representation code. These modifications are referenced in the repository's **modification-ThreadX** file. To evaluate the symbolic execution engine, we provide the intermediate representation (IR) code, which has been compiled from the source code of ThreadX, in the GitHub repository.

A.3 Set-up

A.3.1 Installation

After cloning or downloading the repository, please refer to the "Preparation" section of the `README` files in the corresponding subdirectories (i.e., `symbolic-execution-engine` and `Proof-of-Concept`). All experiments should be conducted within the Docker

image. If you are unable to build the image, we also provide a pre-built Docker image available on both Docker Hub and Zenodo.

A.3.2 Basic Test

If the Docker image is built or downloaded successfully, you can run `./run_docker.sh` in the repository, which will spawn a shell within the container on success. Note that the container will continue running after you exit the shell. You can reattach to the container,or spawn a new shell to create a new container by running `./run_docker.sh` again later. Please be aware that the `./run_docker.sh` command will delete the existing container and create a new one.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Vulnerable system call Detection. Using the symbolic execution engine, we can identify the modification fields and path constraints of the target system calls, enabling us to further determine the vulnerable system calls. This is proven by the experiment (**E1**) whose results are illustrated in §6.2 (**RQ1**, Table 2 and Table 3).

(C2): Efficiency. The symbolic execution engine is efficient. This is proven by the experiment (**E2**) whose results are illustrated in §6.2 (**RQ4**, Table 2).

(C3): Attack Implications. KOM can be launched on various platforms. This is proven by the experiment (**E3**) whose results are illustrated in §6.2 (**RQ5**, Table 6).

A.4.2 Experiments

Please refer to the "Replicating Our Experiments" section of the `README` file in the repository for detailed steps to reproduce our experiments.

We design experiments (**E1-E2**) for the symbolic execution engine to confirm **C1** and **C2**. We design PoCs (**E3**) to confirm **C3**.

(E1): *[Vulnerable System Call Detection.] [5 human-minutes + 10 compute-hours]:* The vulnerable system calls can be evaluated by performing symbolic execution on the system calls of ThreadX. Please refer to the "(E1) Vulnerable System Call Detection" section of the `README` file in the `symbolic-execution-engine` folder of the repository. The results can be found in `M[1,2,3]_vulnerable_system_calls.xlsx`. **C1** is confirmed if the acquired results are consistent with those in Table 2 and Table 3.

(E2): *[Efficiency.] [2 human-minutes + 1 compute-minute]:* The efficiency of the symbolic execution engine is evaluated by checking the execution results (e.g., the consumed time and the number of executed instructions). This experiment should be conducted after **E1**. Please refer to the "(E2) Efficiency" section of the `README` file in the `symbolic-execution-engine` folder of the repository. The results can be found in `symbolic-execution-run-time-evaluation.xlsx`. **C2** is confirmed if the acquired results are comparable with those in Table 2.

(E3): *[Attack Implications] [5 human-minutes + 10 compute-minutes]:* The attack implications are evaluated by performing the PoC experiments on various platforms. Please refer to the `README` file in the `Proof-of-Concept` folder of the repository. **C3** is confirmed if these PoC experiments can be successfully executed.

A.5 Notes on Reusability

The under-constrained symbolic execution engine is designed to identify vulnerable system calls in ThreadX, specifically for KOM attacks. Currently, this tool does not support the latest version of ThreadX or other RTOSs.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Save what must be saved: Secure context switching with Sailor

Neelu S. Kalani *
EPFL & IBM Research — Zurich

Thomas Bourgeat
EPFL

Guerney D. H. Hunt
IBM T. J. Watson Research Center

Wojciech Ozga
IBM Research — Zurich

A Artifact Appendix

A.1 Abstract

We introduce a tool called Sailor that automatically identifies ISA-state that must be swapped across context switches to prevent information leaks or computational integrity breaches. Sailor parses traces generated from an existing tool, Isla, a symbolic execution engine for ISA specifications written in the domain-specific language, Sail. We demonstrate Sailor for the RISC-V architecture. In this work, we extend Isla to automatically generate traces for multiple instructions, in the same format as provided by the RISC-V specification manuals. Based on the identified ISA-state, we automatically generate tests to check if the context switch implementations on existing systems swaps the state correctly or not.

The associated artifacts include:

1. The patch to Isla for generating traces for all instructions in a general-purpose RISC-V processor (rv64gc).
2. The python scripts to compute the ISA-state relevant for context switching. This includes parsing of the Isla traces (ISA-Inspector) and the classifier algorithm (Analyzer), as shown in Figure 3 of the paper.
3. The automatic test generator (also shown in Figure 3) that generates tests to provide experimental evaluation of existing systems.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Executing the artifacts does not impose any risk for the evaluators, since our work is more focused on the defensive side of security using ISA specifications in Sail and the Isla symbolic execution engine to generate insights useful for implementing secure context switching. We do have tests that help us uncover bugs in existing systems. However, these tests don't perform attacks, rather serve as proof-of-concept for information leakage. Further, these tests require testing on the StarFive VisionFive2 board (we provided access to it via ssh) and not on the evaluators machine.

There are no data privacy or other ethical concerns, since we do not handle private or sensitive data in our work.

A.2.2 How to access

The artifact is accessible on GitHub for the purpose of functionality and reproducability checks. (GitHub link: https://github.com/neeluk7/sailor_artifact.git). The artifact is also hosted on Zenodo: https://doi.org/10.5281/zenodo.15613680.

A.2.3 Hardware dependencies

We ran our experiments on a Linux (v6.8.0) Ubuntu 24.04.2 LTS server equipped with Intel Xeon Gold 5520+, with 112 cores and 504 GB RAM. Running our experiments on any machine with fewer resources should not affect reproducability for all experiments except for the Isla trace generation process. The trace generation takes 20 hours on our server, however, it might take more time depending on the machine being evaluated on. We provide pre-generated traces which require about 3 GB of storage and tests to generate full traces and also a subset of traces.

A.2.4 Software dependencies

We have tested the experiments mainly on Linux (reproducing them on a MacOS should be possible, but we recommend the evaluators to use a Linux system if available). The software dependencies are shown in Table 1 and the Makefile has all the required commands to install these.

A.2.5 Benchmarks

None.

*Majority of the work was done while the author was at IBM Research — Zurich.

Software	Version
RISC-V Toolchain	13.3.0
Opam	2.1.5
Rustc	1.86.0
Python3	3.12.3

Table 1: Software dependencies.

A.3 Set-up

A.3.1 Installation

The instructions and commands to install the dependencies and setup the artifact are included in the README.md and the Makefile. This includes setting up the software dependencies mentioned in Table 1 as well as Sail and Isla. Evaluators can follow the steps below. (In case of difficulties,

1. Run `make setup-dependencies` to install all dependencies.
2. Run `make setup-sail` and `make setup-isla`. These commands will apply the patches to Sail and Isla and build Isla.
3. Download the Isla traces into the *isla_traces_dir* directory inside the root directory of the artifact from Google drive (https://drive.google.com/drive/folders/1FI_wnHABUfFjuzru2wMaTpUQf9I1Qw3r?usp=share_link); the traces consume about 3 GB of disk space.

A.3.2 Basic Test

For the basic test, once the installation is complete, Isla's functionality and the parsing of Isla's traces using the python scripts in Sailor must be checked. More specifically, the following two tests must pass.

1. Isla runs correctly after applying the patch with our modifications. For this, run `make isla-traces-test`. It will produce test traces in the *isla_traces_test* directory and use the `diff` command to compare against expected output in *expected_results/isla_test_traces*.
2. Our script parses the Isla traces correctly. Run `make isla-parse-test`, which will generate the output in the *parse_test_output.txt* and compare it against the expected result using `diff`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our framework Sailor, leverages existing tool, Isla, and the machine-readable ISA-specification in Sail, to identify ISA-state that must be swapped during context switching. The framework produces results as shown in Tables 1, 2, and 3 in the paper. We will regenerate these results using experiment (E1).

(C2): Based on the identified results, we perform a security analysis of three existing systems: Keystone, Komodo, and StarFive VF2 (as shown in Table 4 in the paper). We will corroborate these findings using experiment (E2).

A.4.2 Experiments

(E1): *[Produce security-sensitive ISA-state using Sailor] [30 human-minutes + 1 compute-hour]:*
Preparation: The Isla traces must be downloaded and present in the *isla_traces_dir*, before performing this experiment.
Execution: Run the `make run-sailor` command which will parse all the Isla traces and produce the ISA isnights corresponding to Tables 1, 2, and 3 in the paper as well as run the analyzer (classifier algorithm) to generate insights on which state to swap during context switches. This command will also use the `diff` utility to compare against expected results and automatically generate the tests.
Results: CSVs corresponding to Tables 1, 2, and 3:

1. CSVs/csr_(read/write)_access.csv
2. CSVs/csr_footprint_per_instruction_(machine/ supervisor/user).csv
3. CSVs/instruction _access_per_mode.csv

Further, the security-sensitive insights from the analyzer will be generated in `switch-from-U-to-U.csv`. The tests will be generated in the *test-generator/tests* directory.

(E2): *[Security analysis] [30 human-minutes + 30 compute-minutes]:*
Preparation: Part of this step requires the StarFive VisionFive 2 board to run the automatically generated tests. The time to run the actual tests is less than 5 minutes. We provided remote access to our board via ssh for the evaluation.
Execution: The aim of this step is to verify the results reported in Table 4 of the paper. Keystone and Komodo are both included in the repository as submodules for helping with the simple check required to confirm the finding using `grep`. Run the `grep -nriI "senvcfg" .` commands from within the *keystone* and the *serval/monitors/komodo* directories. The expected output of the command is that no match will be found, since these monitor implementations do not correctly configure/swap the `senvcfg` CSR. Next, the tests automatically generated using experiment (E1) should be run on the StarFive VisionFive 2 board. There are screenshots of the scripts run and expected output in the drive link https://drive.google.com/drive/folders/1EDGzDCXFdQ6UgCNybJUyhg6bmKnJuuGY?usp=share_link. As done in the scripts, it is important

to use taskset to run the tests on the same core to observe the findings.
Results: Once the execution is done and produces expected output, the findings reported in Table 4 are verified.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Flexway O-Sort: Enclave-Friendly and Optimal Oblivious Sorting

Tianyao Gu
Carnegie Mellon University
Oblivious Labs Inc.

Yilei Wang
Alibaba Cloud

Afonso Tinoco
Carnegie Mellon University
Oblivious Labs Inc.

Bingnan Chen
HKUST

Ke Yi
HKUST

Elaine Shi
Carnegie Mellon University
Oblivious Labs Inc.

A Artifact Appendix

A.1 Abstract

These artifacts are meant to complement the paper Flexway O-Sort: Enclave-Friendly and Optimal Oblivious Sorting.

Our goal with the artifacts is to provide our opensource implementation of Flexway O-Sort, experiments that allow to easily replicate the results in our paper, and details explanation on how the security goals are achieved by our implementation code. We additionally provide the code of all the baseline / state of the art algorithms used to compare against our implementation.

We provide all our experiment code either as single line commands or as a simple script file to make results simpler to reproduce, and the commands/file should be simple to modify to try new sets of experimental parameters.

Our main goal with the experiments is not to reproduce the exact same numbers as we have in our graphs, as it will vary greatly with hardware used, but to show that the speedup we obtain, as well as the assymptotic behavior is similar to the shown in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. Our code does not have any destructive steps nor does it disable any security mechanisms. We do not export any data and the data used is artificial.

A.2.2 How to access

The code artifacts are available in zenodo: https://zenodo.org/records/14629454

A.2.3 Hardware dependencies

In terms of infrastructure all of our experiments were run on the same machine, with the following specifications:

- Intel Xeon Platinum 8352S processor with 2.2 GHz base frequency
- 1TB DDR4 RAM, 512GB of maximum EPC size.

A.2.4 Software dependencies

Our artifacts are meant to be run under docker, please see the installation steps in order to know how to build and launch the docker container.

A.2.5 Benchmarks

We ran artificial benchmarks on sorting arbitrary arrays comparing with several sorting implementations. And on generating a histogram, initializing an ORAM and load balancing, comparing Flexway O-Sort with bitonic sort.

A.3 Set-up

A.3.1 Installation

First, build the docker image:

Listing 1: build docker images

```
docker build -t cppbuilder:latest
  ./tools/docker/cppbuilder
```

Second, for every test, run docker at the top level of the repo, mounting the ssd for storage and with access to SGX (running with privileged works for this):

Listing 2: start test environment

```
docker run -v /tmp/sortbackend:/ssdmount \
--privileged -it --rm -v $PWD:/builder
  cppbuilder
```

We expect every experiment to be run inside this docker environment.

A.3.2 Basic Test

We include 3 tests here.

(T1) Make sure the code compiles:

Listing 3: compile the code

```
rm -rf build # optional
export CC=/usr/bin/clang
export CXX=/usr/bin/clang++
cmake -B build -G Ninja
  -DCMAKE_BUILD_TYPE=Release
ninja -C build
```

(T2) Make sure the unit tests pass:

Listing 4: run unit tests on the code

```
./build/tests/test_algo
```

(T3) Make sure the test enclave compiles and a basic sorting test runs in HW mode:

Listing 5: compile and run a test enclave

```
cd applications/sorting
./algo_runner.sh
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Flexway O-Sort is an assymptotically optimal and concretely efficient sorting algorithm suitable for implementation in hardware enclaves such as Intel SGX having speedups when compared to other state of the art oblivious sorting algorithms for enclaves.
(E1a) and (E1b) show the speedup of Flexway O-Sort against other oblivious sorting algorithms in an enclave setting, supporting the claim (C1).

(C2): Flexway O-Sort is also optimal when the input size is smaller than the EPC size, and still outperforms other state of the art oblivious sorting algorithms for enclaves.
(E4) Shows the speedup of Flexway O-Sort against other oblivious sorting algorithms when the input size is smaller than the enclave size. From it we can see that Flexway O-Sort also outperforms the other state of the art oblivious sorting algorithms, supporting the claim (C2).

(C3): The shuffling algorithm used in Flexway O-Sort outperform the state of the art oblivious shuffling algorithms, such as Waks-in/Waks-out and OrShuffle, regardless of element and input size.
(E2a) and (E2b) show the speedup of Flexway O-Sort against other oblivious shuffling algorithms in an enclave setting, supporting the claim (C3).

(C4): Using Flexway O-Sort as a replacement to commonly used oblivious sorters, such as bitonic sorter, leads to immediate speedups in various applications.
Experiments (E3) show the speedup for ORAM initialization, histogram generation and database join, when we replaced Flexway O-Sort with an external memory optimized recursive bitonic sorter. The results support the claim (C4).

(C5): Our implementation of Sorting are memory oblivious. We don't have any experiment for this, but we encourage looking at both the algorithm description in the paper and the code to conclude that all branches and memory accesses do not depend on private data.

A.4.2 Experiments

Most of our experiments are run across exponentially increasing array sizes from a few thousand to 10^9 elements. We also vary the size of each element, from 8 bytes to 1 kilobyte.

For all the experiments where data < EPC, we run them through ninja test, for all the experiments where EPC < data, we have a file 'applications/sorting/algo_runner.sh' that launches the experiments and needs to be modified accordingly for each experiment, please see the comments in the file for information regarding what the modifications do, and see the file appplications/sorting/README.md for the parameters used for each figure.

(E1a): *[Sorting, EPC < data] [15 human-minutes + 10 compute-hour]: Fig. 7(a)*
How to: Change the parameters in algo_runner.sh to match the ones for Figure 7(a). Run the script. Collect the points and plot them using a log-log scale.
Preparation: Do the initial setup.
Execution: Change the parameters in algo_runner.sh to match the ones for Figure 7(a). Run the script. Collect the points in some file and plot them using a log-log scale.
The script will generate a file name as:

```
<ALGORITHM_NAME>_<MIN_ELEMENT_SIZE>_
<MAX_ELEMENT_SIZE>_<MIN_ARRAY_SIZE>_
<MAX_ARRAY_SIZE>.out.
```

Each line on the file will have the format:

```
<ARRAY_SIZE> <ELEMENT_SIZE> <TIME in
  seconds>
```

Collect these points in some file and plot them using a log-log scale.
Results: After plotting all the graphs, you should obtain a plot similar to Figure 7a. in the paper. You can see that Flexway O-Sort (KWayButterflySort) outperforms the other oblivious sorting algorithms, and has a slope similar to the one of the non-oblivious merge-sort.

(E1b): *[Sorting EPC < data] [15 human-minutes + 10 compute-hour]: Fig. 7(b)*
How to: Repeat the same steps as in E1a, but with the parameters for Figure 7(b).
Preparation: same as E1a.
Execution: same as E1a.
Results: You should obtain a plot similar to Figure 7b. in the paper. You can see that Flexway O-Sort outperforms the other oblivious sorting algorithms.

(E2a): *[Shuffling, EPC < data, varying array size] [15 human-minutes + 10 compute-hour]: Fig. 10(a)*
How to: Repeat the same steps as in E1a, but with the parameters for Figure 10(a).
Preparation: same as E1a.
Execution: same as E1a.
Results: You should obtain a plot similar to Figure 10a. in the paper. You can see that KWayButterfly Shuffle outperforms the other oblivious shuffling algorithms.

(E2b): *[Shuffling, EPC < data, varying item size] [15 human-minutes + 10 compute-hour]: Fig. 10(b)*
How to: Repeat the same steps as in E1a, but with the parameters for Figure 10(b).
Preparation: same as E1a.
Execution: same as E1a.
Results: You should obtain a plot similar to Figure 10b. in the paper. You can see that KWayButterfly Shuffle outperforms the other oblivious shuffling algorithms.

(E3a): *[Application Benchmarks, EPC < data] [15 human-minutes + 3 compute-hour]: Table 3 (128MB EPC)*
How to: Change the parameters in algo_runner.sh to match the ones for Table 3: Benchmark Results for Different Applications. Run the script. Collect the speedup between running the application with Flexway O-Sort and the optimized recursive bitonic sorter in a table similar to Table 3 in our paper.
Preparation: Do the initial setup.
Execution: Change the parameters in algo_runner.sh to match the ones for Table 3: Benchmark Results for Different Applications. Run the script. Collect the points in a table and see that there is always a speedup when using Flexway O-Sort instead of the bitonic sorter.
The script will generate a file in the same format as in E1a. Each benchmark type will have a different format, printing the elements size, some debug information and the time it took to run the algorithm using Flexway O-Sort and the optimized recursive bitonic sorter.
Results: You can see that by replacing the optimized recursive bitonic sorter with Flexway O-Sort there is always a speedup in the different applications tested.

(E3b): *[Application Benchmarks, EPC > data] [15 human-minutes + 3 compute-hour]: Table 3 (EPC > data)*
How to: Build the project as in (T1) and run the binary in ./build/tests/test_apps. The tests will output the runtime for baseline and our as can be seen in table 3.
Preparation: Do the initial setup and compilation in releases mode (T1).
Execution: Build the project as in (T1) and run the binary ./build/tests/test_apps:
Collect the points in a table and see that there is always a speedup when using Flexway O-Sort instead of the optimized recursive bitonic sorter.
Each benchmark type will have a different format, printing the elements size, some debug information and the time it took to run the algorithm using Flexway O-Sort and the optimized recursive bitonic sorter.
Results: You can see that by replacing the optimized recursive bitonic sorter with Flexway O-Sort there is always a speedup in the different applications tested.

(E4): *[Sorting, EPC > data] [15 human-minutes + 3 compute-hour]: Fig. 6(a)*
How to: Run the ninja tests described bellow for the sorting algorithm, described bellow, which uses the parameters for Figure 6(a).
Preparation: Do the initial setup and ninja build.
Execution: Run the following command:

```
./build/tests/test_basic_perf
  --gtest_filter=*TestSortInternal*
```

This will output the time it took to sort arrays of different sizes using several algorithms, when EPC > data. Collect the points in a file and plot them using a log-log scale, similarly to E1a.
Results: You can see the behavior of the different sorting algorithms when EPC > data. You can conclude that Flexway O-Sort (KWayButterflySort) outperforms the other oblivious sorting algorithms, and has a slope similar to the one of the non-oblivious std::sort.

A.5 Notes on Reusability

We included our code as part of the opensource Oblivious Data Structure Library project on https://github.com/odslib/oblsort. We are actively developing it, both in order to do further reserach in oblivious algorithms as well as in order to provide a way for developers to incorporate oblivious algorithms into enclave code. We made our code to be easy integrable into enclave applications. The several application examples we included are a good starting point on how to use this sorting primitive.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.





USENIX Security '25 Artifact Appendix: Treebeard: A Scalable and Fault Tolerant ORAM Datastore

Amin Setayesh Cheran Mahalingam Emily Chen Sujaya Maiyya
University of Waterloo

A Artifact Appendix

This artifact provides a functional implementation of Treebeard. Treebeard is implemented in Go and includes all the necessary components to demonstrate the core features of the system, including its multilayer architecture (router, stash, and ORAM layers), batching mechanisms, and fault-tolerant coordination using the Raft consensus protocol. The source code is publicly available at: https://figshare.com/articles/software/treebeard_zip/29230676?file=55094552.

A.1 Abstract

This artifact provides a working implementation of Treebeard, a scalable and fault-tolerant ORAM datastore. It includes the full source code, configuration, and instructions for running the system and observing its core functionality.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact does not involve any real user data, sensitive information, or interaction with external systems beyond local or cloud-based test deployments. All experiments are performed using synthetic workloads. There are no known security, privacy, or ethical concerns associated with running or testing this artifact.

A.2.2 How to access

The source code is publicly available at: https://figshare.com/articles/software/treebeard_zip/29230676?file=55094552.

A.2.3 Hardware dependencies

The system can be run on any general-purpose machine. No specialized hardware (such as SGX, GPUs, or TPUs) is required.

A.2.4 Software dependencies

To run the artifact, you need the following installed locally: Go version 1.20 and Ansible. You will also need access to an Ubuntu server. The Ansible scripts provided will automatically install all required packages and configure the environment on the server. No additional manual setup is necessary beyond running the Ansible playbook.

A.2.5 Benchmarks

A sample workload is provided with the artifact to test the functionality of the system. This workload uses a YCSB-style client to issue GET and PUT operations and verify that Treebeard handles requests correctly. No additional datasets or external inputs are required.

A.3 Set-up

A.3.1 Installation

Install Go v1.20 and Ansible using official instructions.

A.3.2 Basic Test

Please follow the instructions under the "Example Experiment" section of the README.md file in the artifact repository. It provides all necessary installation and configuration steps to prepare the environment.

A.4 Evaluation workflow

A.4.1 Major Claims

None.

A.4.2 Experiments

None.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Learning from Functionality Outputs: Private Join and Compute in the Real World

Francesca Falzon
ETH Zürich, Switzerland

Tianxin Tang
Eindhoven University of Technology, Netherlands

A Artifact Appendix

A.1 Abstract

Private Join and Compute (PJC) is a two-party protocol recently proposed by Google for various use-cases, including ad conversion (Asiacrypt 2021) and which generalizes their deployed private set intersection sum (PSI-SUM) protocol (EuroS&P 2020). We analyze the risks associated with the PJC functionality output, and we describe and implement four practical attacks that break the other party's input privacy, and which are able to recover both membership of keys in the intersection and their associated values.

At a high level, this artifact consists of two parts:

1. An implementation of our search tree attack, located in the `search_tree` directory of our codebase.
2. Implementations of our maximum-likelihood estimation attack, the compressed sensing attack, and the discrete fourier transform attack, all of which are located in the `cs_dft_mle` directory.

This artifact appendix describes the necessary steps for reproducing the results presented in Section 7.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This work is compliant with the Usenix ethics guidelines. Our work aligns with the intended use of PJC and therefore does not expose any vulnerabilities of the protocol or underlying cryptographic primitives.

Use of Synthetic Data. Our experiments exclusively use synthetic data generated by sampling numerical values from uniform and normal distributions. The use of synthetic data ensures that no real personal information is utilized or exposed, thereby eliminating any risks of harm to individuals or groups.

Disclosure. We reached out to Lepoint et al. (ASIACRYPT 2021) and the PJC code repository maintainers[1] on 19.11.2024 and 22.11.2024, respectively, to inform them about our analysis and suggest including information about our attacks in the README of the PJC code repository. They agreed to discuss the results internally and have updated their README[2] to include links to the prior attack works and additional information about the potential dangers of allowing multiple protocol invocations on the same or similar input set.

A.2.2 How to access

For this evaluation process, please access the artifact using https://doi.org/10.5281/zenodo.15147738.

A.2.3 Hardware dependencies

This artifact can be run on a commodity laptop with at least 16 GB of RAM and 32 GB of storage.

A.2.4 Software dependencies

This artifact does not require a specific OS but does rely on the proprietary software Gurobi Optimizer. The following steps describe how to obtain Gurobi Optimizer and the free academic license:

1. Go to the "Gurobi User Portal" at https://portal.gurobi.com/iam/login/ and register a new account using your university email address. When prompted, select "Academic".
2. After activating and logging into your academic account, generate your free academic license under the "Licenses" tab by selecting the "WLS Academic" option.
3. Download the generated lisense at https://license.gurobi.com/manager/licenses/
4. Edit the environment/shell path by running:
   `export GRB_LICENSE_FILE=path/to/gurobi.lic`
5. Download the Gurobi Optimizer (version 11.0 or later) that is suitable for your OS. (Tested with Gurobi Optimizer version 11.0.2 build v11.0.2rc0 for mac64[arm].)
6. To verify that Gurobi Optimizer and the license are installed correctly, please run
   `gurobi_cl --license`

[1] https://github.com/google/private-join-and-compute

[2] https://github.com/google/private-join-and-compute/commit/c0cb3f5b5b616caaaef80b7f7e9c335f4fe53ce7

A.2.5 Benchmarks

We do not require any external dataset for our experiments as we only use synthetic data, and we have included the data generation as part of our artifact.

A.3 Set-up

A.3.1 Installation

Ensure you have the Gurobi Optimizer and license installed, as described in Section A.2.4. The following instructions describe how to use a virtual python environment for the artifact evaluation. We tested our artifact using `python 3.10.15`.

1. If you do not have `virtualenv` installed, install it with:
   `pip3 install virtualenv`
2. Create a new virtual environment:
   `virtualenv myenv`
3. Activate the virtual environment:
   - Windows: `myenv\Scripts\activate`
   - MacOS/Linux: `source myenv/bin/activate`
4. Install the dependencies:
   `pip3 install -r requirements.txt`

A.3.2 Basic Test

1. To run a simple functionality test for the search tree attack, run the following command from the top-level directory:
   `python3 search_tree/test.py`
2. For the basic test for CS, DFT and MLE attacks, navigate to `cs_dft_mle/experiment/`, and run the following command:
   `python3 experiment.py ../config/test_non_sparse.json ../config/test_sparse.json`

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): When carrying out the search-tree attack (described in Section 4 of our paper), increasing the partition size generally resulted in fewer queries with some exceptions. Notably, target set size $|T| = 10^3$ and partition set size $\ell = 2^6$ resulted in fewer queries than $\ell = 2^8$ for all values of intersection ratio ρ tested; additionally, when $\ell = 2^{10}$, we have that $\ell \geq |T|$, and thus all values are recovered with exactly one query. Similar trends are observed for $|T| = 10^4$ and $|T| = 10^5$. This is proven by the experiment (E1) described in Section 7.1 of our paper whose results are illustrated in Figure 3.

(C2): We implemented our recovery methods, maximum likelihood estimation (MLE), discrete fourier transform (DFT), and compressed sensing (CS) using the Gurobi Optimizer. We then ran experiments for both noiseless and noisy recoveries using synthetic data with value ranges of $[-100, 100]$ and $[-1000, 1000]$. We report the recovery results over five runs, with statistics for the value range $[-1000, 1000]$ shown in Table 2 (noiseless) and Table 3 (noisy); results for the value range $[-100, 100]$ are presented in Table 4 (noiseless) and Table 5 (noisy) in the Appendix. We describe the steps to reproduce our results in Experiment E2 and briefly describe our observations below.

MLE. We focus on non-sparse recovery using MLE by setting the intersection size equal to the target set size. We observed that even after $0.95n$ queries for n targets, there is still some ℓ_1 loss in value recovery. This loss increases significantly when the value range is expanded to $[-1000, 1000]$ (see Tables 2 and 3). Moreover, adding noise to each query result (i.e., inner product) does not significantly affect the losses.

DFT. Our results indicate that DFT recovery performs very well in certain experiments. It consumes significantly less time compared with other methods. However, it can be unstable and is not robust to noise.

Compressed Sensing (CS). In the noiseless setting, both CS-ℓ_1 and CS-ℓ_2 achieve exact recovery for intersection ratios up to 0.2 with high probability. CS-ℓ_1 is more scalable than CS-ℓ_2; for CS-ℓ_2, we were only able to scale up to $n = 2000$ using Gurobi (with our table reporting $n = 1000$). For noisy recovery, both methods show robustness to truncated Gaussian noise with standard deviation $\sigma \in \{2.5, 5\}$.

A.4.2 Experiments

(E1): Search Tree Attack Experiment [5 human-minutes + 50 compute-minutes (Short Experiment)/7.34 compute-hours (Full Experiment) + 5GB disk]. Time estimates are based on running the experiments on a 16GB Apple M2 Pro.

Preparation: Make sure to follow the setup steps described in Appendix A.3 of this writeup. Before running this experiment, create the directory in which the results will be stored by running the command:
`mkdir results`

Execution: To run the experiments for the search tree attack, run the following commmand from the `search_tree` directory of the repository:
`python3 benchmark.py NUM-RUNS parameter`
where `NUM-RUNS` is a positive integer denoting how many times to run the attack for each parameter setting and `PARAMS` takes on the value of either `part` (which runs an abbreviated set of parameters and which can run in <1hr on a commercial laptop) or `full` (which runs the full set of parameters reported in our paper). For example, the command
`python3 benchmark.py 1 part`

specifies running the benchmarks for the abbreviated set of parameters for 1 run per parameter setting.
Results: To reproduce the graphs, first navigate to `results-part.tex` (if running the abbreviated parameter selection) or `results-full.tex` (if running the full parameter selection) and compile the LaTeX document. These documents pull the data from `results` and print the graphs depicted in Figures 3 and 7.

(E2): Noiseless and noisy recovery using CS, DFT and MLE [5 human-minutes + 4 compute-hours] To save evaluation time, we set the number of iterations to 1:
Preparation: Ensure that you have completed the setup steps described in Appendix A.3. Work within the `cs_dft_mle` directory. Navigate to the `experiment` directory. Remove the `results` directory if it exists due to basic test using `rm results`.
Execution:

1. (15 compute-minutes) Follow the following steps to produce results for target set size $n \in \{10^2, 10^3\}$:
   ```
   python3 experiment.py
   ../config/e1_non_sparse.json
   ../config/e1_sparse.json
   ```
   It outputs a `results` directory containing CSV files that report the results.
2. (3 compute-hours) The following step produces results for $n = 10^4$ (excluding the CS-ℓ_2 for noisy recovery, as explained in the paper):
   ```
   python3 experiment.py
   ../config/e2_non_sparse.json
   ../config/e2_sparse.json --exclude
   ```
   This updates the CSV files in the `results` directory.

Results: To generate the LaTeX tables from the results, run the following command:
```
python3 gen_tex_table.py 1000
```
(Optional) If `pdflatex` is installed, you can preview the generated tables using the following command:
```
pdflatex gen_table_pdf.tex
```
This generates `gen_table_pdf.pdf` that contains two tables similar to Table 2 and Table 3 presented in our paper.
Optional: For Table 4 and Table 5 in the Appendix, which differ in value range from $[-1000, 1000]$ to $[-100, 100]$ and noise magnitude, the evaluation steps are similar. First backup the results directory using
```
mv results results_1000
```
then run the following commands:
```
python3 experiment.py
../config/e3_non_sparse.json
../config/e3_sparse.json
```
and
```
python3 experiment.py
../config/e4_non_sparse.json
../config/e4_sparse.json --exclude
```
and finally
```
python3 gen_tex_table.py 100
```
(Optional) If you have `pdflatex` installed, you can preview the tables using the following command:
```
pdflatex gen_table_pdf.tex
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: ALERT: Machine Learning-Enhanced Risk Estimation for Databases Supporting Encrypted Queries

Longxiang Wang[†,*], Lei Xu[‡,†,*], Yufei Chen[†], Ying Zou[‡], Cong Wang[†]

[†]City University of Hong Kong [‡] Nanjing University of Science and Technology

A Artifact Appendix

A.1 Abstract

The artifact is used to evaluate ALERT in Dynamic Searchable Symmetric Encryption (DSSE) schemes. It includes source code for reproducing ALERT and three baseline LAAs, along with the datasets and scripts required to run the experiments. A comprehensive README file is also included to facilitate easy reproduction of the results.

A.2 Description & Requirements

In this section, we detail the accessibility of our artifact, specify the required hardware and software dependencies, and describe the benchmarks used in our experiments.

Specifically, our artifact contains eight folders: baselines, dataset, final_result, log, model, result, scripts, and src. The "baselines" folder contains code for evaluating results from three baseline LAAs. The "dataset" folder stores preprocessed documents for searchable encryption. The "final_result", "log", "model", and "result" folders contain essential results and data (e.g., trained models, extracted co-occurrence matrices) in the risk assessment process. The "scripts" folder provides scripts for launching risk assessments across different SSE schemes and adversary assumptions. The "src" folder contains ALERT's core source code, including data preprocessing utilities, classifier training modules, and query risk assessment implementations.

A.2.1 Security, privacy, and ethical concerns

Throughout our study, we have conducted a thorough ethical assessment of our research methodology and potential impacts. Our investigation does not involve human participants, personal information, or any sensitive data. The research relies exclusively on three publicly available datasets for experimental evaluation, neither of which contains any personally identifiable information. Therefore, there are no ethical concerns in this paper.

* The first two authors contributed equally to this work.

A.2.2 How to access

Our artifact is available through Zenodo. The artifact can be accessed at https://doi.org/10.5281/zenodo.14726862.

A.2.3 Hardware dependencies

We implemented and evaluated ALERT on a server equipped with two Intel Xeon Platinum 8383C CPUs, eight Nvidia RTX A6000 GPUs, 2TB of RAM, and 12TB of disk storage. While the implementation of ALERT is GPU-based, its main functionality can also be validated by training the model on alternative CPU/GPU configurations.

A.2.4 Software dependencies

Our experiments were primarily conducted on Ubuntu 22.04, though our system is compatible with any operating system that supports Python. For dependency management, we use "Conda" as our environment management system. All required packages and their specific versions are documented in the provided "requirements.txt" file in the published repository.

A.2.5 Benchmarks

In the experiment, we use three widely adopted public datasets from different domains: the Enron email dataset, the NYTimes article dataset, and the Wikipedia encyclopedia entries dataset. All datasets used in our experiments are included in the "dataset" directory of our published artifact. Detailed information about the workloads is presented in Section A.4.2.

A.3 Set-up

This section provides detailed instructions for setting up our experimental environment. The setup utilizes Conda for environment management and includes a verification step to confirm the correct installation of all components.

A.3.1 Installation

To ensure full compatibility with ALERT, we recommend creating a new conda environment with Python 3.9.16:

```
conda create -n alert python=3.9.16
conda activate alert
```

Next, install the required dependencies:

```
pip install -r requirements.txt
```

The experiments can be reproduced with the provided scripts.

A.3.2 Basic Test

We provide a simple test to verify that all dependencies are properly installed. Execute the following commands:

```
cd scripts
bash test_single_risk_assessment.sh
```

If dependencies are correctly installed, a file should appear in the "final_result/test_single_risk_assessment" directory.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): ALERT demonstrates robust risk assessment capabilities across diverse datasets and adversary settings. This is validated through experiments (**E1**) detailed in Sections 5.2.2 and 5.2.3, with results in Figure 4.

(C2): ALERT is able to provide risk assessment maps among keywords and illustrate listed findings: (1) Larger γ generally leads to higher overall leakage risk. (2) An increase in leakage (γ) may introduce noise that obscures distinct patterns, making certain keywords harder to recover. **E2** detailed the experiment steps, corresponding to results in Section 5.2.1 and Figure 3.

(C3): ALERT shows robustness across FP/BP-DSSE schemes (with different λ values). Experimental validation (**E3**) confirms the findings in Section 5.4, with corresponding results visualized in Figure 6.

(C4): Keyword clustering based on volume information is effective. Section 5.3 describes experiment results (**E4**), whose results are plotted in Figure 5.

(C5): Keyword dynamic clustering mechanism (DCM) can significantly reduce risk assessment latency while maintaining risk assessment accuracy. **E5** shows the experiment steps, corresponding to Section 5.3 and Table 1.

(C6): ALERT is the only method to consistently achieve a high query recovery rate with stable convergence under low-latency scenario among the selected risk analysis methods. **E6** demonstrates the experiment steps, corresponding to Section 5.5.1 and Figure 7 in the paper.

(C7): ALERT achieves competitive query recovery performance compared to other LAA methods without time constraints. The validation workflow (**E7**) corresponds to Section 5.5.2 and results in Figure 8 in the paper.

(C8): ALERT exhibits superior recovery performance (median query recovery rates) compared with Jigsaw under similar runtime constraints in large keyword universe sizes scenario. Experimental validation through **E8** corresponds to Section 5.5.3 and Figure 9 in the paper.

(C9): ALERT shows higher median query recovery rates compared with prior alternatives against three padding countermeasures. **E9** specifies experiment workflows, corresponding to Section 5.5.4 and Figure 10.

A.4.2 Experiments

This section provides experimental procedures to validate our major claims. Each experiment description is structured into three main blocks: Preparation, Execution, and Results, ensuring a clear approach to reproducing our findings.

Preparation: Before running each script, change the directory to the "scripts" directory and activate the created conda environment. Set the "if_gpu" field in each script to choose between CPU or GPU training. If using GPU mode, specify the desired GPU deviced with "GPU_NUM" (e.g., "GPU_NUM=0:1" to use GPUs 0 and 1).

(E1): [Query Recovery Performance on Different Datasets and Adversarial Assumptions] [5 human-minutes + 60 compute-hours + 510GB disk]: The experiment performs risk assessments for Enron, NYTimes, and Wikipedia under two adversarial settings.

Execution: To run the experiment:

```
bash main_recovery.sh
```

Results: The results are stored in the "final_result/main_recovery" folder. Each file contains a "cumulative_accuracy" field that records recovery rates for different keyword numbers. The filename indicates the experimental settings: dataset names (Enron/NYTimes/Wiki), adversary type (partial/sample), and adversary assumptions (data partial known rate γ, and data sampling rate α).

(E2): [An Example of Recovery Map of ALERT] [5 human-minutes + 4 compute-hours + 10GB disk]: This experiment demonstrates a visual representation of ALERT (Figure 3 in paper). It's important to note that risk assessment results for concrete queries may exhibit variance across different training sessions due to randomness in training data selection. Therefore, we provide specific training results (probability matrices) for generating the heat map shown in the paper.

Execution: To generate the heatmap:

```
bash test_heatmap.sh
```

Results: The heatmaps generated with different data partial known rates γ are available in "final_result/heatmap". When γ is small, the random selection of the training data may introduce variability in risk assessment outcomes, resulting in figures that deviate from those reported in the paper. For reference,

we provide the original data and heatmaps for reference in "final_result/heatmap/ref".

(E3): [Performance across Forward/Backward Privacy-DSSE] [5 human-minutes + 10 compute-hours + 50GB disk]: This experiment aims to illustrate the effectiveness of ALERT in FP/BP-DSSE schemes.

Execution: We vary data deletion rates λ and show its impact on query recovery rates. To run the experiment:

```
bash test_fpbp.sh
```

Results: The results of the experiment are stored in "final_result/test_fpbp" directory, demonstrating the recovery rates under different data deletion rates λ.

(E4): [Keyword Clustering Results with Fixed Parameters] [5 human-minutes + 1 compute-hour]: This experiment validates the effectiveness of keyword clustering based on volume information (as shown in Figure 5).

Execution: The experiment requires first running the "main_recovery" script to generate the co-occurrence log. For independent execution, we provide a separate script to generate these co-occurrence logs.

```
bash test_dcm_fixed_param.sh
```

Then, keyword clustering results can be generated by executing the provided ipynb file:

```
generate_result_dcm_fixed_param.ipynb
```

Results: The notebook contains three sections, each corresponding to a different subfigure in Figure 5, demonstrating the effects of different keyword numbers (ϑ), sampling rates (α), and clustering thresholds (θ).

(E5): [Performance with/without Dynamic Keyword Clustering Mechanism] [5 human-minutes + 45 compute-hours + 50GB disk]: This experiment evaluates the impact of keyword dynamic clustering mechanism (DCM) on query recovery rates and risk assessment latencies.

Execution: We compare the query recovery rates and program runtime with and without the mechanism. Here is the script how to run the experiment:

```
bash test_dcm.sh
```

Results: The results are stored in "final_result/test_dcm" directory. Each output file contains query recovery rates and program runtime.

(E6): [Comparisons with Prior Alternatives under Low-latency Scenario] [5 human-minutes + 45 compute-hours + 10GB disk]: This experiment aims to compare the effectiveness of ALERT with prior alternatives in the risk assessment scenario.

PS: Although ALERT performs a quick risk assessment, the training process takes about 1-2 hours. For quick validation of the main claims, the parameter "RUN_TIME" can be set to 5 in the scripts of later experiments (**E6, E7, E8, E9**) to ensure minimum iterations required for box plot generation. Note that this simplification may lead to increased variance in the results.

Execution: In this section, we adjust the convergence speed parameters for each method: "RefSpeed" for Jigsaw and RSA, "n_iters" for IHOP, and $\tilde{\beta}$ for ALERT. These adjustments control the program runtime and risk assessment time. To run the experiment:

```
bash test_attacks_low_latency.sh
```

Results: The results are reserved in "final_result/test_attacks_low_latency" folder, where each filename indicates the corresponding dataset, attack method, and desired risk assessment latency. Each file contains data for box plot generation, including minimum value, first quartile (Q1, 25th percentile), median, third quartile (Q3, 75th percentile), maximum value, and the average runtime in seconds.

(E7): [Comparisons with Prior Alternatives without Time Constraints] [5 human-minutes + 50 compute-hours + 10GB disk]: This experiment compares the effectiveness of different methods without runtime constraints.

Execution: We provide the script for the experiment:

```
bash test_attacks_default.sh
```

Results: Correspondingly, the results are stored in "final_result/test_attacks_default". Note that we set $\tilde{\beta} = 1$ to maximize the risk assessment recovery accuracy when operating without time constraints. For additional validation with $\tilde{\beta} = 0.4$ under default settings, run:

```
bash test_attacks_default_extend.sh
```

(E8): [Comparisons in Larger Keyword Universe Sizes under Similar Time Constraints] [5 human-minutes + 90 compute-hours + 30GB disk]: This experiment compares ALERT and Jigsaw in the Wikipedia dataset to evaluate their effectiveness facing larger keyword universe sizes while maintaining comparable execution times.

Execution: To run the experiment:

```
bash test_large_keyword_wiki.sh
```

Results: The results are stored in "final_result/test_large_keyword_wiki", where each filename indicates the dataset and corresponding keyword universe size used in the experiment.

(E9): [Comparisons with Prior Alternatives Against Countermeasures] [5 human-minutes + 90 compute-hours + 30GB disk]: This experiment evaluates ALERT and baseline LAAs against three padding countermeasures.

Execution: To facilitate reproduction, we provide a script that executes the complete evaluation process:

```
bash test_against_countermeasures.sh
```

Results: The experimental results are stored in the "final_result/test_against_countermeasures" directory. Each filename contains the corresponding dataset name, risk analysis method, and applied countermeasure.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Distributed Private Aggregation in Graph Neural Networks

Huanhuan Jia[†]
Southeast University

Yuanbo Zhao[†]
Southeast University

Kai Dong[*]
Southeast University

Zhen Ling
Southeast University

Ming Yang
Southeast University

Junzhou Luo
Southeast University

Xinwen Fu
University of Massachusetts Lowell

A Artifact Appendix

A.1 Abstract

This appendix describes the software artifact that implements the algorithms and evaluations proposed in "Distributed Private Aggregation in Graph Neural Networks". Specifically, the artifact includes the source code for `DPA-GNN`, the datasets used in the experiments, the bash scripts for executing the experiments, and the implementation of the comparison methods. These components are essential for reproducing the experiments (E1) to (E10) described in §7.2 of our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our research uses publicly available graph networks that contain no personal or sensitive data, avoiding privacy and ethical concerns. The artifact operates with standard benchmark datasets, adhering to ethical research standards and avoiding risks related to personal data. Additionally, it does not involve destructive actions or disable security mechanisms, ensuring users' machines and data remain secure.

A.2.2 How to access

The artifact is publicly available at https://doi.org/10.5281/zenodo.14710401.

A.2.3 Hardware dependencies

The artifact evaluation requires the following hardware configuration:

- **CPU:** Experiments use an Intel Xeon Gold 6430 processor (16 vCPUs), but comparable CPUs should also be sufficient.
- **RAM:** The server is equipped with 120 GB of RAM.
- **GPU (optional):** Experiments run on an NVIDIA GeForce RTX 4090 GPU, but similar consumer-grade GPUs are acceptable.
- **Storage:** At least 128 GB of storage is required for datasets, models, and experimental results.

A.2.4 Software dependencies

The artifact requires the following operating system and essential software packages:

- **OS:** Windows, Linux, or macOS.
- **Python Version:** Python 3.9.
- **Packages:**

```
- pytorch=2.0.0
- torch-geometric=2.4.0
- torch-scatter=2.1.2
- torch-sparse=0.6.18
- numpy=1.24.4
- pandas=2.0.3
- scikit-learn=1.3.2
- scipy=1.8.0
```

A.2.5 Benchmarks

Datasets. We conduct experiments on 6 public real-world datasets: Cora, CiteSeer, LastFM, Facebook, Amazon, and Reddit, as shown in §7.1.2 of the paper. These datasets are automatically downloaded during the first run of the script.

Baselines. We consider various baseline methods for comparison, as shown in §7.1.3 of the paper. The code for all baseline methods is provided in the `./baselines` directory. These baselines are categorized into three types: enhanced variants of existing methods, existing privacy preserving GNN methods, and centralized node-level DP methods, as detailed below:

[†]These authors contribute equally to this work.
[*]Corresponding author, email: dk@seu.edu.cn.

Enhanced variants of existing methods: Solitude+RR, LPGNN+RR, RGNN+RR

Existing privacy preserving GNN methods: Solitude, LPGNN, RGNN, DPRR, Blink.

Centralized node-level DP methods: DPSGD, GAP-NDP, PNPiGNNs, DPDGC.

A.3 Set-up

A.3.1 Installation

Please verify that the hardware and software dependencies are satisfied, and download the code from the repository. The environment setup required for this artifact is defined in the `requirements.txt` file. Further details, including the file structure and command-line parameters, are provided in the `README.md` file.

A.3.2 Basic Test

To run the basic functionality test, execute the following command:

```
python train.py --c_rate 0.8
                --layerx 10
                --layery 8
                --epsilon_all 8
                --epsilon_rate 0.05
                --dataset cora
```

This command runs a basic test to construct a `DPA-GNN` using the Cora dataset, with a clipping rate of 0.8, 10 feature aggregation layers, and 8 label aggregation layers, under a total privacy budget of 8 and a feature privacy budget allocation ratio of 0.05.

If everything is functioning correctly, the results can be found in `output_results.txt`.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): `DPA-GNN` achieves superior performance while preserving the privacy of node features, edges, and labels, demonstrating an effective utility-privacy trade-off. This is demonstrated by experiment (E1), as detailed in §7.2.1 of the paper, with results presented in Table 3.

(C2): `DPA-GNN` shows significant advantages when edges are private. This is demonstrated by experiments (E2) to (E5), as detailed in §7.2.2 of the paper, with results presented in Figures 3 to 6.

(C3): `DPA-GNN` achieves performance comparable to, or even surpassing, four competing methods when transitioning from distributed to centralized settings. This is demonstrated by experiment (E6), as detailed in §7.2.3 of the paper, with results shown in Figure 7.

(C4): We conduct experiments to evaluate the effect of hyperparameter settings. This is demonstrated by experiments (E7) to (E10), as detailed in §7.2.4 of the paper, with results presented in Figures 8 to 10 and Table 4.

A.4.2 Experiments

(E1): *[Utility-privacy trade-off] [30 human-minutes + 60 compute-hours]: This experiment evaluates the trade-off between utility and privacy achieved by `DPA-GNN` and the baseline methods across six datasets under varying privacy budget settings. The results are presented in Table 3 of the paper.*

Preparation: Upon the first execution of the code, the datasets corresponding to the command will be automatically downloaded to the `./node-level` directory.

Execution: Run the `run_E1.sh` script to reproduce the results of `DPA-GNN` across all six datasets. We set the total privacy budget ε to $\{2,4,6,8,10\}$, with the node representation privacy budget ε_h set to 5%ε and 20%ε. The optimal values for `c_rate`, `layerx`, and `layery` for each `dataset` are summarized in Table 1.

Table 1: Optimal parameter settings for different datasets.

dataset	c_rate	layerx	layery
cora	0.8	10	8
citeseer	0.8	8	10
facebook	0.7	10	6
lastfm	0.6	6	4
amazon	0.4	10	2
reddit	0.2	8	2

The same `run_E1.sh` script can also be used to reproduce the results of the privacy-enhanced variants of existing methods—Solitude+RR, LPGNN+RR, and RGNN+RR—whose source code is available in the `baselines` directory of our Zenodo repository.

Results: Each experiment is repeated 10 times, and the mean and standard deviation are reported. The results of the experiments are recorded in tabular format and saved to `./output/results_E1.csv` file. Note that due to numeric instability, exact numbers from the paper may be impossible to reproduce.

(E2 to E5): *[Ablation studies] [1 human-hour + 72 compute-hours]: These experiments conduct ablation studies by analyzing scenarios in which only certain data types are private, generating the results presented in Figures 3 to 6 of the paper.*

Execution: Run the scripts from `run_E2.sh` to `run_E5.sh` to reproduce the results of Experiments (E2) through (E5) for `DPA-GNN` and the existing privacy-preserving GNN methods across all six datasets.

The parameter settings for `DPA-GNN` differ slightly across the ablation experiments and are summarized as follows.

In Experiment (E2), both `layerx` and `layery` are set to 2. In Experiment (E4), only `layerx` is set to 2 for large-scale datasets. When features or labels are not considered private, the corresponding privacy budget parameters, `epsilon_x` or `epsilon_y`, are set to `inf`. The value of the `dummy_r` parameter is determined based on the ε_e setting described in the paper. To implement the non-private edge setting in Experiment (E5), incorporate the following parameter into the execution command.

```
1        --execute_dummy False
```

Results: The results of the experiments are recorded in the output files `results_E2.csv` to `results_E5.csv`, located in the `./output` directory. Note that due to numeric instability, exact numbers from the paper may be impossible to reproduce.

(E6): *[Scenario transition] [30 human-minutes + 12 compute-hours]: This experiment compares the performance of DPA-GNN with baselines when transitioning from distributed to centralized settings and generates the results presented in Figure 6 of the paper.*

Execution: DPA-GNN operates without any changes to the model architecture or parameter settings when transitioning from distributed to centralized environments. Run the `run_E6.sh` script to reproduce the results of Experiments (E6) for both `DPA-GNN` and the enhanced variants of existing methods.

Results: The experimental results are logged in the `./output/results_E6.csv` file. Note that due to numeric instability, exact numbers from the paper may be impossible to reproduce.

(E7 to E10): *[Effect of parameters] [1 human-hour + 96 compute-hours]: These experiments evaluate the effect of parameters, including the number of layers l_h and l_y, clipping rate, dummy parameter, and selective parameter, generating the results presented in Figures 8 to 10 and Table 4 of the paper.*

Execution: Run the `run_E7.sh` script to reproduce the results of Experiment (E7). This experiment requires modifying the execution command parameters, as shown below:

```
1        --layerx {LAYERX}
2        --layery {LAYERY}
```

Run `run_E8.sh` to reproduce the results of Experiment (E8) by modifying the parameter as shown below:

```
1        --c_rate {C_RATE}
```

Run `run_E9.sh` to reproduce the results of Experiment (E9) by adding the following parameter:

```
1        --dummy_r {DUMMY_R}
```

Run `run_E10.sh` to reproduce the results of Experiment (E10) by adding the following parameters:

```
1        --num_shares {NUM_SHARES}
2        --num_parties {NUM_PARTIES}
```

Results: The results of the experiments are recorded in the output files `results_E7.csv` through `results_E10.csv`, located in the `./output` directory. Note that due to numeric instability, exact numbers from the paper may be impossible to reproduce.

A.5 Notes on Reusability

To use the artifact on other graph datasets, ensure that the dataset is pre-processed into the appropriate format. You can easily add new data loaders to accommodate new datasets.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Suda: An Efficient and Secure Unbalanced Data Alignment Framework for Vertical Privacy-Preserving Machine Learning

Lushan Song
Fudan University & ByteDance

Qizhi Zhang
ByteDance

Yu Lin
ByteDance

Haoyu Niu
Fudan University

Daode Zhang
ByteDance

Zheng Qu
Fudan University

Weili Han
Fudan University

Jue Hong
ByteDance

Quanwei Cai
ByteDance

Ye Wu
ByteDance

A Artifact Appendix

A.1 Abstract

In this paper, we propose `Suda`, an efficient and secure unbalanced data alignment framework for vertical privacy-preserving machine learning (VPPML). `Suda` efficiently, directly, and exclusively outputs data shares in the intersection without expensive secure shuffle operations. Consequently, `Suda` efficiently and seamlessly aligns with secure training in VPPML.

This artifact is a C++ implementation of the protocols presented in this paper. It contains the source code and scripts needed to reproduce our experimental results in this paper. Specifically, we provide: (1) Source code for the functionalities of the secure unbalanced data alignment and batch privacy information retrieve (PIR). (2) Scripts for obtaining the experimental results in our paper.

A.2 Description & Requirements

`Suda` enables a server P_S that holds larger data with size N and a client that holds smaller data with size n ($n \ll N$) to achieve secure data alignment efficiently. After executing `Suda`, P_S and P_C can obtain data shares in the intersection without data shares outside the intersection. Then, these two parties could use the data shares as input to train a machine learning model securely.

In addition, we provide an efficient implementation of the batch PIR protocol. This protocol enables a client P_C to retrieve results corresponding to its batch of queries from the data of a server P_S.

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns associated with this artifact.

A.2.2 How to access

The artifact is open-sourced both at Zenodo https://zenodo.org/records/15109398 and in the Git repository https://github.com/sls33/Suda.

A.2.3 Hardware dependencies

Our experiments were run on a Linux server equipped with an Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz and 720GB of RAM. Please make sure that the machine has at least 500 GB of RAM to obtain the experimental results in our paper.

A.2.4 Software dependencies

We recommend you install the following software dependencies:

- Debian GNU/Linux 10 (buster) or greater.
- Python 3.7 or greater.
- gcc 11.5.0 or greater.
- clang 18.1.8 or greater.

A.2.5 Benchmarks

We employ two datasets as follows to evaluate `Suda`'s performance over public datasets.

- SVHN[1], which contains 73257 training samples and 26032 test samples, each represented as a 32 × 32 RGB image.

[1] http://ufldl.stanford.edu/housenumbers/

- Character Font Images[2], which contains 832670 samples, for each sample we select a 20×20 grayscale image and 8 additional features.

Besides, we use logistic regression to evaluate the performance of secure training using the outputs of secure unbalanced data alignment.

A.3 Set-up

Follow the "Requirement" of README documentation in the Git repository https://github.com/sls33/Suda, which will guide you through setting up the required workspace.

A.3.1 Installation

We provide instructions on how to install the dependencies and necessary configuration steps in the README documentation of https://github.com/sls33/Suda.

A.3.2 Basic Test

After installing all the dependencies and third-party libraries, you can run the command `$ ./build/bin/psi_to_share_test 20 1024 100 0 test_ps.txt & ./build/bin/psi_to_share_test 20 1024 100 1 test_pc.txt` to run a simple functionality test. In this command, 20 refers to the larger data size $N = 2^{20}$, 1024 refers to the smaller data size $n = 1024$, and 100 refers to the feature dimensions $m = 100$. The results are stored in the files "test_ps.txt" and "test_pc.txt" as follows.
test_ps.txt:

```
host_log_n_data=20
batch_size=1024
feature_num=100
party_id=0
Performance:
communication size (send + recv): 91951.4 KBytes
mem usage of server: 2317.16MB
total time of server: 191306
```

test_pc.txt:

```
host_log_n_data=20
batch_size=1024
feature_num=100
party_id=1
Performance:
communication size (send + recv): 91951.4 KBytes
mem usage of client: 293.742MB
total time of client: 191728
```

[2] https://archive.ics.uci.edu/dataset/417/character+font+images

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Suda can efficiently achieve secure unbalanced data alignment between two parties. This is proven by the experiment (E1). The experimental results are described in Section 5.2 of the paper and illustrated in Tables 1, 2, 3, and 4.
(C2): Suda's outputs enhance the efficiency of secure training in VPPML. Therefore, Suda seamlessly aligns secure training in VPPML. This is proven by the experiment (E2). The experimental results are described in Section 5.2.1 of the paper and illustrated in Table 1.
(C3): Suda can efficiently achieve batch PIR. This is proven by the experiment (E3). The experimental results are described in Section 5.3 of the paper and illustrated in Table 5.

A.4.2 Experiments

(E1): *[Efficiency of Secure Unbalanced Data Alignment] [30 human-minutes + 25 compute-hour + 100GB disk + 300GB RAM]: This experiment evaluates the efficiency of secure unbalanced data alignment in Suda over different data settings, including public datasets, varied data sizes, varied feature dimensions and varied intersection size.*
Preparation: Build the environment as described in the following link https://github.com/sls33/Suda.
Execution: Follow the scripts of "Efficiency of secure unbalanced data alignment" in the following link https://github.com/sls33/Suda to run the experiments and retrieve the results. Note that the source code of the baseline CPSI is in https://github.com/Visa-Research/volepsi.git. If you want to obtain their experiment results, you can follow the instructions in their repository.
Results: The experimental results would be stored in text files. The README documentation of https://github.com/sls33/Suda provides more details. This experiment supports claim (C1).
(E2): *[Efficiency of Secure Training] [10 human-minutes + 10 compute-hour + 100GB disk + 16GB RAM]: This experiment evaluates the efficiency of secure training using the outputs of secure unbalanced training alignment in Suda.*
Preparation: Build the environment as described in the following link https://github.com/sls33/Suda.
Execution: Follow the scripts of "Efficiency of secure training" in the following link https://github.com/sls33/Suda to run the experiments and retrieve the results.
Results: The experimental results would be stored in text files. The README documentation of https://

github.com/sls33/Suda provides more details. This experiment supports claim (C2).

(E3): *[Efficiency of Batch PIR] [10 human-minutes + 20 compute-hour + 100GB disk + 500GB RAM]: This experiment evaluates the efficiency of batch PIR protocol in* `Suda`.

Preparation: Build the environment as described in the following link https://github.com/sls33/Suda.

Execution: Follow the scripts of "Efficiency of batch PIR" in the following link https://github.com/sls33/Suda to run the experiments and retrieve the results. Note that the source code of the baseline `PIRANA` is in https://github.com/zju-abclab/PIRANA. If you want to obtain their experiment results, you can follow the instructions in their repository.

Results: The experimental results would be stored in text files. The README documentation of https://github.com/sls33/Suda provides more details. This experiment supports claim (C3).

A.5 Notes on Reusability

You can configure the larger data size N, the smaller data size n, and the feature dimensions m in the command as shown in Section A.3.2.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Assuring Certified Database Utility in Privacy-Preserving Database Fingerprinting

Mingyang Song[1], Zhongyun Hua[1], Yifeng Zheng[2], Tao Xiang[3], Guoai Xu[1], and Xingliang Yuan[4]

[1]*School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen*
[2]*Department of Electrical and Electronic Engineering, The Hong Kong Polytechnic University*
[3]*College of Computer Science, Chongqing University*
[4]*School of Computing and Information Systems, University of Melbourne*

A Artifact Appendix

A.1 Abstract

We provide the source code used for experimental evaluations, along with a roadmap for deploying and executing the source code. The open source code includes three parts: (i) the source code for dataset extraction and encoding, (ii) the source code of UtiliClear, and (iii) the source code for robustness and utility testing. The experiments in the paper involve evaluating efficiency, fingerprint robustness, and database utility. The efficiency evaluation results can be obtained during the execution of the source code of UtiliClear, while the results of fingerprint robustness and database utility can be obtained by executing the third part of the artifact.

A.2 Description & Requirements

This section outlines the required setups for executing the artifact. Before running the code, please set up the execution environment according to the following hardware and software configurations. Afterward, follow the execution steps presented in the artifact precisely to run the source code.

A.2.1 Security, privacy, and ethical concerns

The artifact involves neither malicious nor destructive operations. It poses no risk to machines security, data privacy or others ethical concerns for evaluators while executing the artifact.

A.2.2 How to access

The artifact and detailed deployment roadmap for executing UtiliClear are now available at Zenodo.

A.2.3 Hardware dependencies

Any testing environment is sufficient to reproduce only the overall efficiency trends. However, to reproduce the similar efficiency results in the paper, two desktops, each equipped with a 24-core Intel Core i9-13900 processor and 16GB of memory, are required to execute the source codes for the database owner and recipient, respectively.

A.2.4 Software dependencies

The source code rely on standard cryptographic libraries, including, OpenSSL, PBC, GMP, and MySQL. The dataset used for evaluation is available for download at https://nijianmo.github.io/amazon/index.html.

A.2.5 Benchmarks

We evaluate UtiliClear using the Amazon Review Data.

A.3 Set-up

Install Visual Studio 2015 and PyCharm 2021.3.1.

A.3.1 Installation

Before running the programs, the libraries of OpenSSL-3.0.4-Win32, PBC-0.5.14-Win32, GMP-Win32, and MySQL-5.1.51-Win32 should be installed and included in the programs

A.3.2 Basic Test

To run a simple functionality test, one can use the encoded subset (186 MB) along with its corresponding dictionary files provided in the artifact. With this small dataset, the first part (i.e., dataset extraction and encoding) can be skipped. Directly use the small subset as the input to execute the second part

(i.e., the source code of UtiliClear). Before executing the programs, we recommend setting the number of group to 400, the bit-length of insignificant bits to 1, and the number of modifiable bits to 8. Then run the source code following the execution roadmap provided by the readme file of the artifact. Both the code files 'Com_Verify_Recipient.cpp' and 'InsignificantBits_Verify3_Recipient.cpp' will output 'success', indicating that software components are correctly utilized and functioning fine.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): During the preprocessing and verification phases, the time costs for both the recipient and the database owner increase with the number of groups and the number of insignificant bits. In contrast, during the fingerprint embedding and extraction phases, the database owner's computation time increases with the number of insignificant bits but remains unaffected by the number of groups. This is demonstrated by the experiment (E1), whose results are reported in Figures 7-8.

(C2): During the preprocessing phase, the communication overhead increase with the number of groups and the number of insignificant bits. In contrast, during the verification phase, the communication overhead increases with the number of insignificant bits. This is demonstrated by the experiment (E1), whose results are reported in Figure 9.

(C3): More groups cause smaller data sizes per group, reducing memory requirements for both the database owner and the recipient. This is demonstrated by the experiment (E1), whose results are reported in Table 2.

(C4): UtiliClear achieves comparable fingerprint robustness and database utility with the state-of-the-art scheme. This is demonstrated by the experiment (E2), whose results are reported in Figures 12-14.

A.4.2 Experiments

(E1): It takes approximately 10 hours to obtain a single experimental result and several days are needed to obtain all the efficiency evaluation results presented in Section 6.2 using the suggested hardware configuration above.
How to: Use the first part of the source code to process the database, then take the encoded database as input to execute the second prat of the source code.
Preparation: Before executing the code, set the parameters (e.g., the ratio of removed records, the ratio of added records, and the ratio of flipping bits) in code files.
Execution: Use the first part of the source code to process the database and obtain encoded databases. Following the detailed roadmap described in the open artifact, execute the source code of UtiliClear to obtain the experimental results.
Results: Track the execution time of the code to obtain the computational time consumption. For communication consumption, monitor the files exchanged between the recipient and the database owner. To observe memory cost during code execution using Visual Studio.

(E2): It takes approximately two days to obtain the experimental results of fingerprint robustness. Four days are needed to obtain the results of database utility.
How to: Use the outputs (i.e., fingerprinted databases) of the source code of UtiliClear to evaluate database utility and fingerprint robustness by executing the third part of the source code.
Preparation: Before executing the code, set the parameters in each code file and ensure the consistency of parameter settings in all code files.
Execution: Following the detailed roadmap described in the open artifact, execute the third part code files to test the fingerprint robustness and database utility.
Results: Obtain the fingerprint robustness results by comparing the fingerprint extracted from the databases under various types of attack with the original fingerprint. To obtain the database query result, decode the fingerprinted database and import the decoded dataset and the original dataset into the MySQL database, and then test the SQL query results. One code file (i.e., 'Classifier_Training.py') is provided to train and evaluate the classifier's accuracy.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: "Shechi: A Secure Distributed Computation Compiler Based on Multiparty Homomorphic Encryption"

Haris Smajlović* University of Victoria
David Froelicher* MIT
Ariya Shajii Exaloop Inc.
Bonnie Berger MIT
Hyunghoon Cho Yale University
Ibrahim Numanagić University of Victoria

A Artifact Appendix

A.1 Abstract

Shechi is a Pythonic framework for high-performance secure distributed computing. Its artifact includes installation, setup, and performance comparison against the other frameworks for secure distributed computing. The comparison is done through four sets of experiments: (i) micro-benchmarks, (ii) basic workflows, (iii) complex workflows, (iv) scalability tests, and (v) ablation study. In the following sections, we provide detailed instructions on how to run each experiment.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no security nor ethical risks while executing this artifact.

A.2.2 How to access

Access Shechi's artifact at https://zenodo.org/records/14868541.

A.2.3 Hardware dependencies

Original Shechi experiments were evaluated in a distributed setup, each party on a different machine with 12-core Intel i7-8700 CPUs (3.20GHz), 64 GB RAM, all connected via a LAN network with 1 Gb/s bandwidth and 0.5 ms latency. For easier artifact evaluation, we have enabled all experiments to be run on a single machine, each party as a separate process communicating over UNIX sockets. The online run, however, is still possible.

Any machine with at least 64 GB RAM will suffice for most experiments. The only experiments that require more RAM (around 200 GB) are the scalability tests that involve more than 3 parties to run on a single machine (as a rough estimate, each party consumes around 20 GB of RAM for the heaviest experiments).

*Co-first authors.

A.2.4 Software dependencies

Shechi runs only on Linux at the moment (any maintained distribution). For reference, our experiments were run on CentOS Stream 9.

A.2.5 Benchmarks

We enabled all experiments to be executed on simulated datasets (generated at random in each run and simulating data similar to the lung cancer dataset from dbGaP (accession: phs000716.v1.p1)).

A.3 Set-up

A.3.1 Installation

We suggest following the installation instructions in the `README.md` in the artifact. The same instructions are provided here for completeness.

Install Codon first:
```
mkdir $HOME/.codon && curl \
-L https://github.com/exaloop/codon/releases/\
download/v0.17.0/codon-$(uname -s | awk \
'{print tolower($0)}')-$(uname -m).tar.gz | \
tar zxvf - -C $HOME/.codon --strip-components=1
```
And then install Sequre/Shechi:
```
curl \
-L https://github.com/0xTCG/sequre/releases/\
download/v0.0.20-alpha/sequre-$(uname -s | \
awk '{print tolower($0)}')-$(uname \
-m).tar.gz | tar zxvf - -C \
$HOME/.codon/lib/codon/plugins
```
Finally, add an alias for the sequre command (make sure it is a one-liner in the terminal):
```
alias sequre="find . -name 'sock.*' -exec rm {} \; && CODON_DEBUG=lt $HOME/.codon/bin/codon run \
--disable-opt="core-pythonic-list-addition-opt" \
-plugin sequre"
```

A.3.2 Troubleshooting

Some Shechi experiments require Python. If Shechi fails to automatically link this dependency, make sure to export path

to `libpython.so` shared library to `CODON_PYTHON` environment variable.

A.3.3 Basic Test

Run `sequre examples/local_run.codon` within the artifact. This will run the simple multiplication example in a simulated network of 3 parties. If the output contains the `MHE initialize` log for each party (CP), then Shechi has been successfully installed.

A.4 Evaluation workflow

Please note that all the commands below are also available in `USENIX25_README.md` in the root directory of the artifact.

A.4.1 Major Claims

(C1): The performance of Shechi's low-level operations is on par with the state-of-the-art frameworks and libraries. This is proven by the experiment (E1) described in Section 10.2.1, the results of which are reported in Table 2.

(C2): Shechi is up to 80× faster than the state-of-the-art homomorphic encryption frameworks when computing basic workflows like L2 distance and matrix multiplication. This is proven by the experiment (E2) described in Section 10.2.2, which is illustrated in Figure 7 (left).

(C3): Shechi is up to 15× faster than the state-of-the-art frameworks when computing complex, large-scale workflows such as Kinship estimation based on Genotypes, Principal Components Analysis, and Genome-Wisde Association Study. This is proven by the experiment (E3) described in Section 10.2.3, which is illustrated in Figure 7 (right).

(C4): Shechi's performance scales linearly, while the performance of Secure Multiparty Computation-based framework Sequre scales quadratically when increasing the number of computing parties. This is proven by the experiment (E4) described in Section 10.3, which is illustrated in Figure 8.

(C5): Shechi's ablation study shows that Aggregation and Encoding optimization is essential for good performance. This is proven by the experiment (E5) described in Section 10.4, which is illustrated in Figure 9 (right).

A.4.2 Experiments

(E1): [Micro-benchmarks] [5 human-minutes + 10 compute-minutes + 5GB RAM]: experiment evaluates the runtime of low-level operations in Shechi, Sequre, SEAL, MP-SPDZ, and Lattigo and demonstrates that Shechi's low-level functionalities are on par with the other frameworks.
Preparation: Make sure to run Shechi and Sequre experiments from the root directory of the artifact.
Execution: Shechi and Sequre: `sequre -release scripts/invoke.codon run-benchmarks --local --jit --lattiseq --mpc --mhe`. SEAL: `docker run --rm --privileged hsmile/seal:bench`. MP-SPDZ: `docker run --rm --privileged hsmile/mpspdz:bench`. Lattigo: `docker run -it --rm --privileged hsmile/lattigo:micro`.
Results: Inspect the `stdout` for reported runtimes for each benchmark and benchmarked procedure. The results should reflect the results reported in Table 2.

(E2): [Basic workflows] [5 human-minutes + 1.5 compute-hour + 10GB RAM]: experiment evaluates the runtime and network consumption of computing the L2 distance and matrix multiplication in Shechi, Sequre, HEFactory, EVA, MP-SPDZ, and Lattigo. It demonstrates that Shechi is constantly faster than other HE-based frameworks and consumes moderate bandwidth, as presented in Figure 7 (left).
Preparation: Make sure to run Shechi and Sequre experiments from the root directory of the artifact.
Execution: Shechi and Sequre: `sequre -release scripts/invoke.codon run-benchmarks --local --jit --stdlib-builtin`. HEFactory: `docker run --rm --privileged hsmile/hefactory:latest`. EVA: `docker run --rm --privileged hsmile/eva:bench`. MP-SPDZ: `docker run --rm --privileged hsmile/mpspdz:bench`. Lattigo: `docker run -it --rm --privileged hsmile/lattigo:bench`.
Results: Inspect the `stdout` for reported runtimes and network statistics for each benchmark and benchmarked procedure. Please note that HEFactory and EVA will not provide any network statistics. This is because they are executed in a centralized (non-distributed) fashion, under the assumption that all parties will encrypt and share their data beforehand. The results should reflect the results illustrated in Figure 7 (left).

(E3): [Complex workflows] [5 human-minutes + 18 compute-hours + 60GB RAM]: experiment evaluates the runtime and network consumption of computing the Kinship estimation based on Genotype, Principal Components Analysis, and Genome-Wisde Association Study in Shechi, Sequre, and Lattigo. It demonstrates that Shechi is constantly faster than other frameworks and consumes less bandwidth, as presented in Figure 7 (right).
Preparation: Make sure to run Shechi and Sequre experiments from the root directory of the artifact.
Execution: Shechi and Sequre: `sequre -release scripts/invoke.codon run-benchmarks --local --jit --king --pca --gwas-without-norm`. Lattigo Kinship: `docker run -it --rm --privileged`

`hsmile/lattigo:king`. Lattigo GWAS (with PCA): `docker run -it --rm --privileged hsmile/lattigo:gwas`.

Results: Inspect the `stdout` for each benchmark's reported runtimes and network statistics at the end of the benchmarked procedure. Inspect the `results` directory to check the accuracy, or run `python scripts/accuracy.py` in the root directory of the artifact. Please note that the accuracy of Sequre KING may be off target. This is due to the SMC overflow that can happen for some randomly generated input. The results should reflect the results illustrated in Figure 7 (right).

(E4): [Scalability] [5 human-minutes + 8 compute-hours + 200GB RAM]: experiment evaluates the runtime and network consumption of computing the Principal Components Analysis in three different network setups: 2, 4, and 8 computing parties. It demonstrates that Shechi scales linearly as opposed to Sequre's quadratic performance drop when increasing the number of computing parties, as presented in Figure 8.

Preparation: Navigate to the root directory of the artifact.

Execution: 2-party setup: `SEQURE_CP_COUNT=3 sequre -release scripts/invoke.codon run-benchmarks --local --jit --pca`. 4-party setup: `SEQURE_CP_COUNT=5 sequre -release scripts/invoke.codon run-benchmarks --local --jit --pca`. 8-party setup: `SEQURE_CP_COUNT=9 sequre -release scripts/invoke.codon run-benchmarks --local --jit --pca`.

Results: Inspect the `stdout` for reported runtimes and network statistics. The results should reflect the results illustrated in Figure 8.

(E5): [Ablations] [1 human-minute + 40 compute-minutes + 5GB disk]: experiment evaluates the small ablation study for all possible optimization setups (with each of the three main optimizations either turned on or off). It demonstrates that the Aggregation and Encoding optimization (`enc` in `stdout`) is detrimental to performance and more effective than Maximizing efficient plaintext operations (`pri` in `stdout`) and Minimizing multiplication redundancy (`fac` in `stdout`), as illustrated in Figure 9 (right).

Preparation: Navigate to the root directory of the artifact.

Execution: Run `sequre -release scripts/invoke.codon run-benchmarks --local --jit --ablation`.

Results: Inspect the `stdout` for reported runtimes and network statistics. The results should reflect the results illustrated in Figure 9 (right).

A.5 Notes on Reusability

Shechi is a programming framework with Pythonic domain-specific language and, as such, is not restricted to specific experiments. It is intended for building arbitrary secure distributed applications, and we encourage the users to inspect the examples in `applications` and `stdlib/sequre/stdlib` directories for more examples of Shechi. We also plan to publish a comprehensive documentation and tutorial for Shechi in the near future.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Practically Secure Honey Password Vaults: New Design and New Evaluation against Online Guessing

Haibo Cheng[1], Fugeng Huang[1], Jiahong Yang[1], Wenting Li[2], Ping Wang[1]
[1]Peking University [2]Beijing Institute of Graphic Communication

A Artifact Appendix

A.1 Abstract

This paper evaluates the distinguishability and practical security of existing honey password vault schemes using a large-scale dataset. We also propose a novel scheme based on the Transformer model. Our analysis shows that existing schemes are highly vulnerable to distinguishability attacks, with text classification algorithms (especially pre-trained models) achieving 83.75%–95.79% accuracy. In contrast, our Transformer model generates more plausible decoy vaults, limiting attack accuracy to no more than 64.35%. For practical security against online guessing, our model excels, with an average of only 0.51 accounts cracked per 1,000 attempts, which can be further reduced to 0.11 with two simple measures.

A.2 Description & Requirements

This artifact provides both the source code and a comprehensive large-scale dataset for honey vault research. The code includes implementations of three honey vault schemes (Golla et al., Cheng et al., and ours), existing and proposed attack algorithms, and security evaluations with accompanying plotting scripts. The dataset consists of 7 million password vaults.

A.2.1 Security, Privacy, and Ethical Concerns

Our artifact does not perform any harmful operations on the host machines, nor does it disable any security mechanisms during execution. To protect user privacy, the dataset has been sanitized to remove usernames and directly identifiable information, and is provided in encrypted form with restricted access.

A.2.2 How to Access

Both the source code and dataset are hosted on Zenodo. The source code is publicly available at https://doi.org/10.5281/zenodo.15612143, while the dataset is accessible at https://doi.org/10.5281/zenodo.15646753 under restricted access.

A.2.3 Hardware Dependencies

This artifact requires an NVIDIA GPU with at least 24GB VRAM and CUDA 11.8 support (e.g., NVIDIA RTX 3090). A minimum of 80 GB of system RAM and 50 GB of disk storage are also necessary. While no specific CPU model is strictly mandated, a CPU with at least 20 cores is recommended.

A.2.4 Software Dependencies

Our experiments were conducted and tested on Ubuntu Linux 22.04, utilizing CUDA 11.8, Anaconda, and Python 3.10. All additional software dependencies can be installed during the setup phase.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

1. Download and extract the code and dataset archives from Zenodo (see A.2.2). Place the three files of the full dataset into the `src/honeyvault/data` directory within the extracted artifact.

2. Navigate to the extracted artifact directory. Execute the following commands to set up a Conda environment and install necessary dependencies:

```
conda create -n py310 python=3.10 -y
conda activate py310
pip install -r req_clean.txt
pip install "protobuf>=3.20.0,<4.0.0"
```

   You may encounter warnings about dependency conflicts; please disregard them, as the artifact has been tested with these specific dependency versions.

3. Generate a Wolfram Alpha API key by following the instructions at https://reference.wolfram.com/language/WolframClientForPython/docpages/basic_usages.html#generate-a-secured-authentication-key. Insert this key into line 17 of src/honeyvault/CONFIG_RCDF.py. This key is essential for the vault model fitting process within the scheme proposed by Cheng et al. (2021).

A.3.2 Basic Test

Upon successful installation, verify the artifact's basic functionality by executing the following command:

```
python src/honeyvault/__main__.py
```

This command initiates a default quick-start workflow using a small synthetic dataset. The workflow encompasses training for three honeyvault models, attack model training, online attack evaluation, and result plotting. Expect this entire process to take approximately one hour.

Upon successful execution, the terminal output should display messages similar to `The merged PDF is saved to x/xx/xxx.pdf`. For verification, manually inspect one of the generated PDF plots to confirm its general resemblance to Figure 6 in the paper. Note that the exact curve shapes may vary.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): For traditional evaluation of honeyvault schemes against distinguishing attacks, our DeBERTa-based attack achieves the highest accuracy among both existing and our proposed attacks. This is demonstrated by Experiment (E2) (Section 6.2) and depicted in Figure 5 and Table 3.

(C2): Our scheme exhibits superior resistance against distinguishing attacks. This is demonstrated by Experiment (E2) (Section 6.2) and depicted in Figure 5 and Table 3.

(C3): For practical evaluation of honeyvault schemes against online guessing attacks, our DeBERTaBase attack achieves the best performance. This is demonstrated by Experiment (E1) (Section 6.3) and depicted in Figures 6-14 and Table 7.

(C4): For practical evaluation of honeyvault schemes against online guessing attacks, our scheme achieves the best security. This is demonstrated by Experiment (E1) (Section 6.3) and depicted in Figures 6-14 and Table 7.

(C5): Our two simple measures significantly enhance the security of all three honeyvault schemes against online guessing. This is demonstrated by Experiment (E1) (Section 6.3) and depicted in Table 5 and Table 7.

A.4.2 Experiments

It is strongly recommended to execute the experiments sequentially as listed below, as the initial experiment (E1) automates the training of honeyvault and attack models.

(E1): *[Online Guessing] [5 human-minutes + 3 compute-weeks]:*

Preparation: Adjust the configuration to match the paper's experimental setting. In src/honeyvault/CONFIG_Online_quick_start.py, comment out the `QuickStart` block and uncomment the `Paper` block. For deep learning-based attacks, only the optimal one (DeBERTaBase) is executed by default; alternatively, modify the `DL_model` variable in src/honeyvault/__main__.py:57 to include other deep learning-based attacks.

Execution: Execute the following command:

```
python src/honeyvault/__main__.py
```

Results: Results will be saved in the src/honeyvault/attack/leaky_detection/leak_pic directory. This directory will contain multiple subdirectories, each named to indicate different experimental conditions:

- A name containing `honey_num_0` indicates Measure II is disabled, while `honey_num_2` indicates it is enabled.
- A name including `LIMIT_WEB=False` signifies that Measure I is not applied, whereas `LIMIT_WEB=True` signifies its application.
- A name including `ENHANCE_MPW=False` implies that a strong master password is not enforced, while `ENHANCE_MPW=True` implies its enforcement.
- A name containing `defender_attack` indicates the ideal case, while `defender_attack_practical` indicates the practical case.

Within the deepest subdirectory, attack curves are saved in three PDF files. Files named with `2016`, `2021`, and `transformer` correspond to the schemes by Golla et al., Cheng et al., and our proposed scheme, respectively. The precise average number of cracked accounts under 10, 100, 1000 login attempts are saved in JSON files named `max_y_with_pos_2016.json`, `max_y_with_pos_2021.json`, and `max_y_with_pos_transformer.json` for each scheme. The correspondence between figures and tables in the paper and their respective result files is detailed in Table 1.

(E2): *[Distinguishing Attacks] [5 human-minutes + 2 compute-hours]:*

Preparation: Adjust the configuration to match the paper's experimental setting. In src/honeyvault/CONFIG_RCDF.py, comment out the `Quick_start` block and uncomment the `Paper Config` block (around line 25). Similar to E1, only the

optimal deep learning-based attack (DeBERTaBase) is run by default; modify the `DL_model` variable in `src/honeyvault/attack/__main__.py:319` to include other deep learning-based attacks if desired.
Execution: Execute the following commands:

```
cd src/honeyvault
python -m attack
python draw_picture_final_v2.py
```

Results: Results will be saved in the `src/honeyvault/result/final` directory. RCDFs are saved as three PDF files. Files named with `2016`, `2021`, and `transformer` correspond to the schemes by Golla et al., Cheng et al., and our proposed scheme, respectively. The accuracy α or AUC of distinguishing attacks against honey vault schemes are saved in `700w_average_ranks.csv`. The correspondence between figures and tables in the paper and their respective result files is also detailed in Table 1.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

Table 1: Summary of the major claims and experiments.

Claim	Exp.	Figure or table in the paper	Corresponding result files
C3, C4, C5	E1	Figure 6	`$PATH_E1/defender_attack/honey_num_0_attack_op/ENHANCE_MPW=False_LIMIT_WEB=False/3_attack_curves_p0_*.pdf`
		Table 4 and Table 5: None	`$PATH_E1/defender_attack/honey_num_0_attack_op/ENHANCE_MPW=False_LIMIT_WEB=False/max_y_with_pos_*.json`
		Figure 7	`$PATH_E1/defender_attack_practical/honey_num_2_attack_op/ENHANCE_MPW=False_LIMIT_WEB=True/3_attack_curves_p0_*.pdf`
		Tables 5 and 7: M1 & M2 (Practical Case)	`$PATH_E1/defender_attack_practical/honey_num_2_attack_op/ENHANCE_MPW=False_LIMIT_WEB=True/max_y_with_pos_*.json`
		Figure 8	`$PATH_E1/defender_attack/honey_num_0_attack_op/ENHANCE_MPW=False_LIMIT_WEB=True/3_attack_curves_p0_*.pdf`
		Tables 5 and 7: M1	`$PATH_E1/defender_attack/honey_num_0_attack_op/ENHANCE_MPW=False_LIMIT_WEB=True/max_y_with_pos_*.json`
		Figure 9	`$PATH_E1/defender_attack/honey_num_2_attack_op/ENHANCE_MPW=False_LIMIT_WEB=False/3_attack_curves_p0_*.pdf`
		Tables 5 and 7: M2 (Ideal case)	`$PATH_E1/defender_attack/honey_num_2_attack_op/ENHANCE_MPW=False_LIMIT_WEB=False/max_y_with_pos_*.json`
		Figure 10	`$PATH_E1/defender_attack_practical/honey_num_2_attack_op/ENHANCE_MPW=False_LIMIT_WEB=False/3_attack_curves_p0_*.pdf`
		Tables 5 and 7: M2 (Practical case)	`$PATH_E1/defender_attack_practical/honey_num_2_attack_op/ENHANCE_MPW=False_LIMIT_WEB=False/max_y_with_pos_*.json`
		Figure 11	`$PATH_E1/defender_attack/honey_num_0_attack_op/ENHANCE_MPW=True_LIMIT_WEB=False/3_attack_curves_p0_*.pdf`
		Tables 5 and 7: MPW	`$PATH_E1/defender_attack/honey_num_0_attack_op/ENHANCE_MPW=True_LIMIT_WEB=False/max_y_with_pos_*.json`
		Figure 12	`$PATH_E1/defender_attack/honey_num_2_attack_op/ENHANCE_MPW=False_LIMIT_WEB=True/3_attack_curves_p0_*.pdf`
		Tables 5 and 7: M1 & M2 (Ideal Case)	`$PATH_E1/defender_attack/honey_num_2_attack_op/ENHANCE_MPW=False_LIMIT_WEB=True/max_y_with_pos_*.json`
		Figure 13	`$PATH_E1/defender_attack/honey_num_2_attack_op/ENHANCE_MPW=True_LIMIT_WEB=True/3_attack_curves_p0_*.pdf`
		Tables 5 and 7: M1 & M2 (Ideal Case) & MPW	`$PATH_E1/defender_attack/honey_num_2_attack_op/ENHANCE_MPW=True_LIMIT_WEB=True/max_y_with_pos_*.json`
		Figure 14	`$PATH_E1/defender_attack_practical/honey_num_2_attack_op/ENHANCE_MPW=True_LIMIT_WEB=True/3_attack_curves_p0_*.pdf`
		Tables 5 and 7: M1 & M2 (Practical Case) & MPW	`$PATH_E1/defender_attack_practical/honey_num_2_attack_op/ENHANCE_MPW=True_LIMIT_WEB=True/max_y_with_pos_*.json`
C1, C2	E2	Figure 5	`src/honeyvault/result/final/700w_*_attack_result.pdf`
		Table 3	`src/honeyvault/result/final/700w_average_ranks.csv`

* The variable `$PATH_E1` expands to the path `src/honeyvault/attack/leaky_detection/leak_pic`.





USENIX Security '25 Artifact Appendix: CertPHash: Towards Certified Perceptual Hashing via Robust Training

Yuchen Yang, Qichang Liu,* Christopher Brix[1], Huan Zhang[2], and Yinzhi Cao
{yc.yang, qliu85, yinzhi.cao}@jhu.edu,
brix@cs.rwth-aachen.de, huan@huan-zhang.com
The Johns Hopkins University, [1]RWTH Aachen University, [2]UIUC

A Artifact Appendix

A.1 Abstract

CertPhash is the first certified perceptual hashing (PHash) system with robust training. CertPhash includes three different optimization terms: anti-evasion, anti-collision, and functionality. The anti-evasion term establishes an upper bound on the hash deviation caused by input perturbations, the anti-collision term sets a lower bound on the distance between a perturbed hash and those from other inputs, and the functionality term ensures that the system remains reliable and effective throughout robust training. This artifact includes the source code, dataset, and models, along with instructions to set up the environment, implement, and evaluate CertPHash.

A.2 Description & Requirements

A.2.1 How to access

The code and scripts used in our paper are available in the following GitHub repository, at commit `59ed6e6441e1ba06d09631ec900799eb204079ef`: https://github.com/Yuchen413/CertPhash/tree/main. It is also available at the Zenodo stable URL https://zenodo.org/records/14740844, with the `latest` release tag.

A.2.2 Hardware dependencies

Our artifacts require a Linux machine with 64GB of RAM and a GPU with 40 GB of graphics memory. We use an NVIDIA A100-PCIE-40GB GPU with CUDA Version 12.1 and Driver Version 530.30.02.

A.2.3 Software dependencies

We use Python 3.10 and Conda for package management. The required dependencies can be installed using:

```
conda create -n certphash python=3.10 -y
conda activate certphash
pip install -r requirements.txt
```

Additionally, auto_LiRPA dependencies must be installed:

```
git clone git@github.com:Verified-Intelligence/
    auto_LiRPA.git
cd auto_LiRPA
python setup.py install
```

*This work was done when Qichang Liu was a summer intern at JHU.

A.2.4 Artifact hierarchy

The artifact includes the following three folders:

- `generate_phash`: PHash generation using existing algorithms.
- `train_verify`: Train and verify the CertPhash.
- `attack`: Functionality and empirical attack evaluations.

A.2.5 Datasets

We use datasets COCO, MNIST, CelebA, and NSFW-56K for training and evaluation. Due to ethical concerns, we cannot provide a direct download link for NSFW-56K, please follow the steps in the README of our GitHub repository. The other datasets can be downloaded at this link, which includes images and perceptual hashes used in our experiments. After downloading, unzip the file, name the folder as `data` and replace the existing `train_verify/data` folder with the downloaded one.

Then perform the following procedure for the datasets you need:

- **COCO**: Download images from this ADDITIONAL link and unzip them as `coco100x100`. Place the folder under `train_verify/data`.
- **MNIST**: Ensure a folder named `mnist` containing images and hashes is present.
- **CelebA**: Ensure a folder named `celaba_random` is present.

Please double check that the downloaded data files follow the structure as specified in `./train\_verify/data/put\_data\_here\_follow\_this.txt`:

```
-coco100x100
    -.jpg
-coco100x100_val
    -.jpg
-coco-train.csv
-coco-val.csv
-mnist
    -testing
        -.jpg
    -training
        -.jpg
    -mnist_test.csv
    -mnist_train.csv
-celeba_random
    -.jpg
```

A.2.6 Models

Our certified robust trained models, adversarial trained models and non-robust trained models have been placed in https://drive.google.com/drive/folders/1b7RbO-uDdlvsxgsxE4H-tjrdGVx7pVRu?usp=sharing In order to use our pretrained models, download the folder named `saved_models` and replace the existing `./train_verify/saved_models` with the downloaded version.

A.3 Set-up

A.3.1 Installation

The following steps set up the Conda environment for our Github repository:

```
conda create -n certphash python=3.10 -y
conda activate certphash
pip install -r requirements.txt
```

A.3.2 Basic Test

To verify installation:

```
bash ./train_verify/test_env.sh
```

The expected output for `./train_verify/test_env.sh` should be similar to `./train_verify/test/train_log.txt`, a.k.a. something like the following:

```
Epoch 1, learning rate [0.0005], dir
    one_epoch_test
[ 1]: eps=0.00000048 active=0.3465 inactive
    =0.6532 Loss=0.4681 Rob_Loss=0.4681 Err
    =1.0000 Rob_Err=1.0000 L_tightness=0.5494
    L_relu=0.0023 L_std=0.8852 loss_reg=0.5518
    grad_norm=15.2293 wnorm=13.1858 Time=0.0517
...
Test without loss fusion
[ 1]: eps=0.00000048 active=0.3509 inactive
    =0.6491 Loss=0.4642 Rob_Loss=0.4642 Err
    =1.0000 Rob_Err=1.0000 L_tightness=0.0000
    L_relu=0.0000 L_std=0.8729 loss_reg=0.0000
    wnorm=14.8170 Time=0.0218
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): CertPHash maintains hash functionality across different datasets, achieving high ROC-AUC scores. This is proven by experiments (E1-1, E1-2) described in our paper's Section 6.1 whose results are illustrated in Figure 2.

(C2): CertPHash maintains certified robustness against evasion and collision attacks, achieving high Certified No Evasion Rates and Certified No Collision Rates. This can be verified through the experiments (E2-1, E2-2) in Table 2 and Table 3 described in our paper's Section 6.2.

A.4.2 Experiments

(E1-1) Functionality Evaluation for Trained PHash Models

Preparation: Install dependencies and datasets.

Execution: Navigate to `attack/` and run the following scripts. An example is provided below.

```
python benign0_func_check.py --dataset='
    coco_val' --target='
    photodna_nn_cert_ep1' --model='../
    train_verify/saved_models/
    coco_photodna_ep1/ckpt_best.pth'
```

The expected output of the console contains the names of all the transformations:

```
Original: 100%|--------------------| 1/1
    [00:00<00:00, 22310.13it/s]
Rotate: 100%|--------------------| 15/15
    [00:00<00:00, 32099.27it/s]
...
```

It will take around five minutes to calculate the PHash from different levels of different transformations. The generated hashes will be saved under folder `./attack/func_logs/coco_val_photodna_nn_cert_ep1`.

Then calculate the ROC-AUC via:

```
python benign0_func_AUC.py --dataset='
    coco_val' --target='
    photodna_nn_cert_ep1'
```

The expected output from the console contains ROC-AUC of transformations tested in our papers RQ1. For instance, the output for the Hue transformation should be as follows:

```
...
hue: -180     -150     -120     -90
    -60      -30      0        30       60
          90       120      150
ROC AUC: 1.0000  1.0000   1.0000   1.0000
    1.0000   1.0000   1.0000   1.0000
    1.0000   1.0000   1.0000   1.0000
Mean ROC AUC: 1.0000, Std ROC AUC: 0.0000
...
```

The full results will be saved in `./attack/func_logs/coco_val_photodna_nn_cert_ep1/coco_val_results.txt`. These results should align with our paper's RQ1.

(E1-2) Certified Robust PHash Training:

Preparation: Install dependencies and datasets.

Execution: Instead of using our trained model, you can also train models from scratch. Navigate to `train_verify` folder and run `train.py` with the robust training configuration file. An example of robust training with a perturbation epsilon of 0.0078 on COCO dataset is provided below.

```
python train.py --method=fast --config=
    config/coco.crown-ibp.json \
--eps=0.0078 --dir=saved_models/
    coco_photodna_ep2 \
--scheduler_opts=start=2,length=80 \
--lr-decay-milestones=120,140 --lr-
    decay-factor=0.2 \
--num-epochs=160 --model='resnet_v5' --
    lr=5e-4
```

Results: The robustly trained model will be saved under `saved_models/`. You can later use this model for verification.

(E2-1): Certified No Evasion Rate (CNER) Verification:

Preparation: Install dependencies and datasets.

Execution: Run `verify_evasion.py` with appropriate parameters. See the below example with verifying a model trained with $\varepsilon = 8/255$ under verifying noises $\varepsilon = 8/255$:

```
python verify_evasion.py --data=
    coco \
--epsilon=0.0312 --model='
    saved_models/coco_photodna_ep8
    /ckpt_best.pth'
```

Results: Compare the computed CNER values with those reported in the paper.

(E2-2) Certified No Collision Rate (CNCR) Verification:

Preparation: Install dependencies and datasets.

Execution: Run `verify_preimage.py`. An example is provided below.

```
python verify_preimage.py
```

Results: Compare the CNCR values with those reported in the paper.

A.5 Notes on Reusability

CertPHash can be adapted to other datasets and hashing models with minor modifications to the data-loading code. Additionally, we provide the implementation and evaluation scripts for our non-robust and adversarial-robust trained PHash model, as well as existing PHash systems, including PhotoDNA, PDQ, and NeuralHash. (Note that we cannot directly provide the extracted algorithms or models due to copyright restrictions.) More details can be found in the README of our GitHub repository.

A.6 Version

This submission follows the LaTeX template for Artifact Evaluation V20231005. Further details can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Red Bleed: A Pragmatic Near-Infrared Presentation Attack on Facial Biometric Authentication Systems

Bowen Hu
School of Electrical and Electronic Engineering
Nanyang Technological University

Kuo Wang
School of Electrical and Electronic Engineering
Nanyang Technological University

Chip Hong Chang
School of Electrical and Electronic Engineering
Nanyang Technological University

A Artifact Appendix

A.1 Abstract

Our work demonstrates a practical attack against Windows Hello face recognition under its default configuration. The underlying vulnerability, designated CVE-2025-26644, was patched on April 9, 2025. All systems running Windows 10 or Windows 11 versions prior to 24H2 are susceptible to this exploit.

We provide comprehensive documentation of the hardware design and the attack procedure using our assembled module. Additionally, our research introduces a cross-spectral face generation model, with full source code and pretrained checkpoints publicly released. To support reproducibility, we have also prepared a Google Colab demo to facilitate hands-on experimentation.

A.2 Description & Requirements

Our work requires specialized hardware—a self-assembled near-infrared (NIR) LCD. The bill of materials (BOM) is provided in the main text, while the printed circuit board (PCB) files, 3D-printed enclosure design, and detailed assembly instructions are available on Zenodo. In addition to the NIR LCD, the attack setup requires: 1) A target machine running Windows 10 or Windows 11 with the Windows Hello facial recognition module in its default configuration. 2) An HDMI media source (e.g., another computer) to supply video content to the NIR LCD. Due to the time and cost associated with hardware setup, we have also prepared a web-based demo for evaluation.

The cross-spectral face generation model can be executed on any machine or platform capable of running PyTorch-based deep learning models. To train the model, a publicly available dataset is required, along with at least one pair of near-infrared (NIR) and visible (VIS) images from the target subject. For ease of use, we provide a Google Colab notebook to facilitate the testing process.

A.2.1 Security, privacy, and ethical concerns

Our work does not involve any destructive modifications to the target system. The only ethical consideration is that the test subject must consent to providing their facial biometric data to unlock the registered Windows device.

A.2.2 How to access

We prepared a hardware demo of unlocking the target machine using the Windows Hello module at `http://16.176.135.204/`. Also, we prepare a Google Colab cross-spectral face generation demo in `https://colab.research.google.com/drive/14pasM4oz7O_vQSFr_6-wM_9WUELhKjeo?usp=sharing`.

The above-mentioned link is provided solely for evaluation convenience. Readers can independently assemble the NIR display and reproduce the experiments locally using the materials and instructions we have released.

A.2.3 Hardware dependencies

We do not have hardware requirement in the weblink demo. If the readers want to reproduce the experiment locally, the hardware requires the following items:

- **NIR LCD**: The self-assembled NIR LCD functions as a standard HDMI display, with all technical details—including design files and assembly instructions—available in our paper and Zenodo submission.
- **Target Windows machine**: The target device must be a Windows machine running Windows 10 or Windows 11, with any version earlier than 24H2 without the patch released on Apr 9, 2025.

- **Windows Hello module**: This work requires a Windows Hello module, which can be either integrated into the Windows laptop or desktop, or connected externally via USB.
- **OmniVision OV9281 monochrome camera module**: This module is used for NIR face image sample collection.
- **Raspberry Pi 4 Model B with camera module 3 12MP**: This module is used for VIS face image sample collection.
- **A GPU machine**: A separate machine is required to run the cross-spectral face generation experiments. In our setup, we used a server equipped with an Intel Xeon w7-3465X CPU (56 cores), 512 GB RAM, and an NVIDIA RTX 4090 GPU with 24 GB of memory. However, such high CPU and memory resources are not essential—the primary requirement is a GPU with at least 24 GB of memory for model training. CPU and system memory usage remain minimal during the process.

A.2.4 Software dependencies

There is no software requirement in this demo. If the readers want to train with their own face biometric data, the software requires the following items:

- **DeepFaceLab**: We download the Windows release from https://github.com/iperov/DeepFaceLab?utm_source=chatgpt.com. This model is used for the face region of interest (ROI) crop and face swapping after the cross-spectral generation.
- **CASIA NIR-VIS 2.0 Face Database**: this publicly available dataset is required for training. It can be downloaded from http://www.cbsr.ia.ac.cn/english/NIR-VIS-2.0-Database.html.

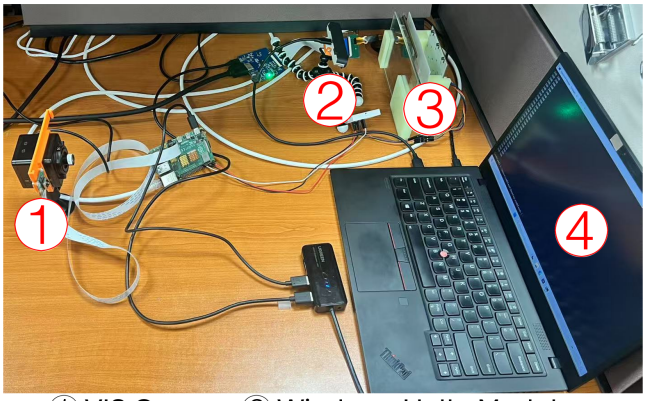
A.2.5 Benchmarks

None.

A.3 Set-up

If not assembling the hardware, the remote demo can be accessed from http://16.176.135.204/. Our web demo is shown in Figure 1. The Raspberry Pi module is used for the Windows machine control and sending back the captured frames. We display the registered subject NIR video on the 850nm LCD in front of the Windows Hello module. The target machine installed Windows 11 23H2 with version released before Apr 9, 2025.

No setup requirement for the Colab. The demo can run directly. https://colab.research.google.com/drive/14pasM4oz7O_vQSFr_6-wM_9WUELhKjeo?usp=sharing.



① VIS Camera ② Windows Hello Module
③ NIR LCD ④ Target Windows Machine

Figure 1: Web demo setting.

A.3.1 Installation

None.

A.3.2 Basic Test

Hardware Web Demo: Press the "Turn On Light" Button to turn on the 850 nm backlight. Press "Send Space" button to start the face unlock. It is able to enter the system bypassing Windows Hello.
Cross-spectral Face Generation: Run the whole Colab Notebook. It is able to generate NIR image samples from the provided VIS image sample.

A.4 Evaluation workflow

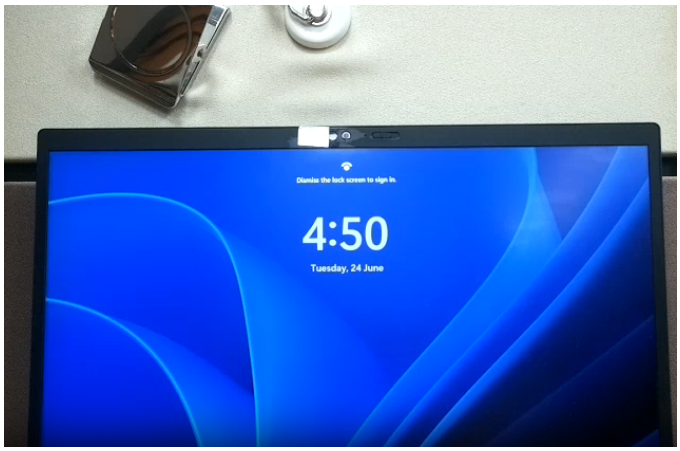
A.4.1 Major Claims

(C1): Using the NIR display is able to bypass the Windows Hello face recognition.
(C2): Using the Variational Autoencoder (VAE) is able to generate NIR image samples from the VIS image samples.

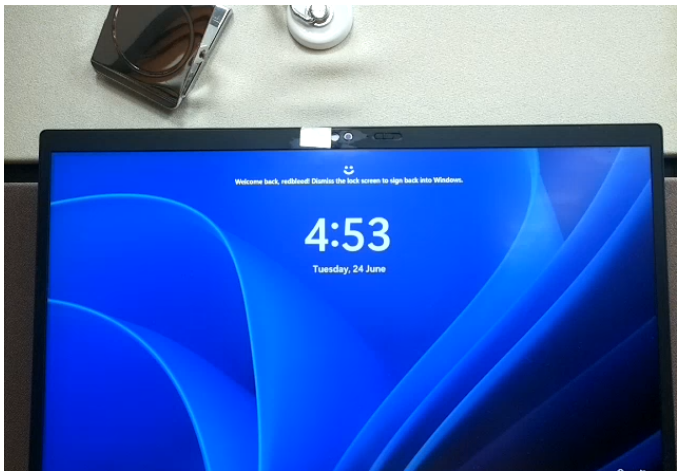
A.4.2 Experiments

(E1): *[Hardware Web Demo] [10 human-minutes]:* There may be few seconds delay in the live video stream due to the network.
How to:

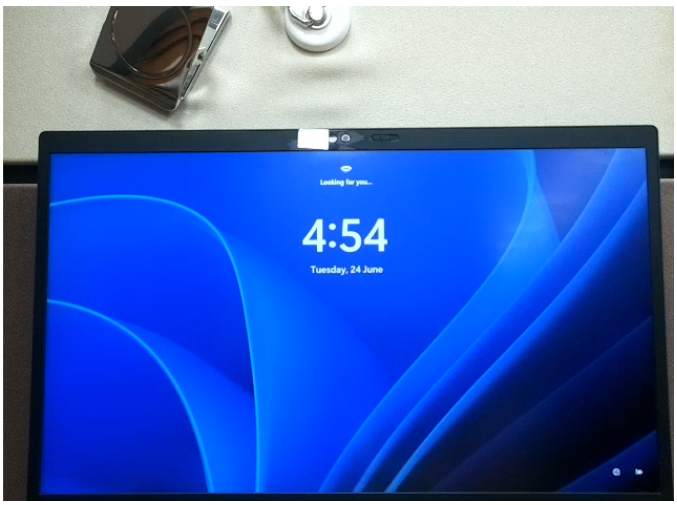
1. The web page is opened, and the video stream shows the target Windows 11 device. It should be in the stand by mode as shown in Figure 2a.
2. Click **"Turn On Light"** to turn on the 850 nm light on the NIR LCD for one minute. Press **"Send Space"** to activate the face unlock routine. The screen will show the machine is ready to unlock as



(a) The target machine in the "standby" mode.



(b) The target machine in the "unlock ready" mode.



(c) The target machine in the "looking for face" mode.

shown in Figure 2b. If we click the **"Send Space"** again, it will enter the system.

3. Click **"Send Lock (Win+L)"** to lock the screen. Click the **"Turn Off Light"** button, now the Windows Hello module cannot find the face, it will keep looking for the face as shown in Figure 2c. If we turn on the light again, it will be back in the unlock ready state again.
4. If it fails to unlock and enters the input pin mode, click the **"Send PIN"** to unlock the system, and Lock again by clicking **"Send Lock"**.

Results: It successfully bypasses Windows Hello facial recognition and gains access to the system by displaying a crafted NIR video on the NIR screen.

(E2): *[Cross-spectral Face Generation.] [20 human-minutes]:*

How to: Executing the entire notebook will automatically initiate the download of sample data and pretrained models, and subsequently perform the complete experimental procedure without further user intervention.

Results: Generating NIR image samples from the provided VIS images.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Oblivious Digital Tokens

Mihael Liskij, ETH Zurich
Xuhua Ding, Singapore Management University
Gene Tsudik, UC Irvine
David Basin, ETH Zurich

A Artifact Appendix

A.1 Abstract

Our artifact consists of two parts:

(A1): A Tamarin model of the protocol we use to prove that our protocol satisfies the binding integrity property. It consists of an ODT protocol model and a proof that the model satisfies this property.

(A2): The ODT protocol prototype implementation. It consists of an ODT client implementation that uses Intel SGX with OpenSSL, an ODT server implementation based on OpenSSL, various auxiliary scripts, and the raw measurement results. We also provide a Dockerfile and Docker image for easy setup and testing.

Each component comes with a `README.md` document that describes how to setup and use the component.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

One of our scripts for artifact (A2) temporarily disables address space layout randomization (ASLR) until it is completed. However, in case the script is interrupted, ASLR might remain disabled until the computer is restarted. To remedy this, we provide a script `enable_aslr.sh` that the evaluators can use at the end of their evaluation of artifact (A2) to ensure that ASLR is enabled. In case you use the Docker deployment of artefact (A2), this is no longer a potential problem.

A.2.2 How to access

The latest artifact version can be found on GitHub[1] and on Zenodo[2].

A.2.3 Hardware dependencies

Artifact (A1) does not require special hardware and the complete proof can be computed and inspected on commodity hardware with 8 GB of RAM and 10 GB of swap space. It takes up to 30 minutes to compute the proof and up to 30 minutes to open it for inspection, with an `Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz` CPU. The model and the proof itself need only 1 MB of storage space.

Artifact (A2) requires an Intel processor with at least SGX1 capabilities; SGX2 capabilities are not required. We test and develop artifact (A2) on the same device as artifact (A1). After installation, the artifact needs between 5 and 6 GB of space on the filesystem.

A.2.4 Software dependencies

Artifact (A1) requires the Tamarin prover, which is available for MacOS, Linux, and Windows.

Artifact (A2) is developed for Linux. It requires the Intel SGX SDK driver, Intel SGX SDK library, Intel SGX PSW library, the Intel SGX SSL library, and an OpenSSL installation. We provide instructions on how to setup the build tools and the Intel SGX SDK driver. For the rest of the dependencies, we provide a script that allows a user to interactively install them, and a Docker image (and corresponding Dockerfile) where the dependencies are installed automatically. For both options, the dependencies are fixed to specific versions and should continue to install and work in the future. Note that Docker containers use the SGX driver from the host kernel. Therefore, it is mandatory to install the Intel SGX SDK driver on the host system.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The artifacts can be downloaded from Github and Zotero, as specified in Section A.2.2. Artifact (A1) is stored in the `tamarin_proof` directory and artifact (A2) is stored in the `ODT_implementation` directory.

Artifact (A1) does not need to be installed, but requires the Tamarin prover to inspect it. Installation instructions can be found on the following URL: `https://tamarin-prover.com/manual/master/book/002_installation.html`.

Artifact (A2) must be installed, either manually or through Docker, and requires the Intel SGX driver. The Intel SGX driver is included in the newer Linux kernels, but it must be explicitly installed for older ones. Therefore, we recommend

[1] https://github.com/Anonymous-Usenix-25/Oblivious-Digital-Tokens/releases/tag/usenix

[2] https://zenodo.org/records/14765576

that evaluators update their kernel if possible. For hardware that supports only SGX1, the kernel driver seems to not work, so one must manually install the older driver according to the instructions found at: https://github.com/intel/linux-sgx-driver. For the remaining installation steps, we provide a `setup.sh` script in the `scripts` directory of artifact (A2) that performs the installation process. The installation process takes around 90 minutes on our hardware and asks for superuser privileges for certain installation steps. If an error occurs, the script will be interrupted and you should investigate the cause of the error. As aforementioned, we provide a Docker image as an alternative to running the `setup.sh` script. Once the Docker image is downloaded, or regenerated from the Dockerfile, one can start an interactive session in the image to obtain a test-ready environment.

Note that we provide extended installation and usage instructions for both artifacts in the `README.md` files found in the respective artifacts' directories.

A.3.2 Basic Test

For artifact (A1), calling `tamarin-prover --version` should show no errors. If no errors are shown, then Tamarin prover is installed correctly.

For artifact (A2), the script `heap_verification.sh`, found in the `scripts` directory, performs an ODT verification based on heap measurements. It configures and recompiles the ODT client and server for heap verification and performs an ODT verification. This process takes around ten minutes to complete. If the heap verification was performed successfully, the expected last line of the output is: "ODT verification success".

We also provide a verification test based on stack measurements. To perform it, we provide the script `stack_verification.sh` that configures and recompiles the ODT client and server for stack verification, disables ASLR, and performs an ODT verification. This process takes around ten minutes to complete. If the stack verification was performed successfully, the expected last line of the output is: "ODT verification success" (and an additional message that ASLR was enabled again). The script asks for superuser privileges to disable and enable ASLR.

A.4 Evaluation workflow

A.4.1 Major Claims

We make the following major claims in the paper:

(C1): Our ODT protocol, shown in Figure 5, satisfies binding integrity as described in Section 7.3. This is proven by experiment (E1).

(C2): Our ODT server prototype incurs a slowdown of less than 1 millisecond compared to a normal OpenSSL server. The experiment (E2) is described in Section 6.3.2 and the results are reported in Table 2.

(C3): Our O-TEE implementation incurs a slowdown of around 144 millisecond compared to a normal OpenSSL client. The experiment (E3) is described in Section 6.3.2 and the results are reported in Table 3.

A.4.2 Experiments

(E1): [Tamarin proof] [15 human-minutes + 60 compute-minutes + 1MB disk]:

Preparation: Install Tamarin prover as described in the installation instructions found at https://tamarin-prover.com/manual/master/book/002_installation.html.

Execution: To explore the finished proof, launch Tamarin in interactive mode by calling it with `tamarin-prover interactive .` (the dot is part of the command) in the `tamarin_proof` directory and open the Tamarin Web interface at http://127.0.0.1:3001.

To redo the whole proof, we provide a partially proven ODT model in the `partial-proof.spthy` file, an oracle file `myoracle.py`, and a script `prove.sh`. The `partial-proof.spthy` file contains a proof of the last lemma that we construct by choosing the proof steps ourselves. We do this manually because Tamarin's autoprover has a tendency to get stuck in an infinite loop. Tamarin can prove all other lemmas with the help of the `myoracle.py` file. You can use the `prove.sh` script to invoke Tamarin on the `partial-proof.spthy` file and have it construct the complete proof `proof2.spthy`.

Upon successful completion of the script, Tamarin should say that all lemmas are verified. You can find the script output in `proof2.log`. The last 43 lines of `proof2.log` should roughly match the content of the `expected_log_output.txt` file. Note that `partial-proof.spthy` and `proof2.spthy` are the output of Tamarin's proof process and are not designed for direct inspection. You must use Tamarin's web interface to inspect them.

Results: In the Tamarin Web interface, you can explore any completed proof, e.g., `proof.spthy` or `proof2.spthy`. After clicking on a proof, and after around 30 minutes, the interface will show the names of all successfully proven lemmas in green on the left side. If all the lemmas are green, then this serves as proof for claim (C1). Some lines of the last proof should be red since the last lemma tests for the existence of a solution, so a single complete trace is enough to prove the lemma.

(E2) and (E3): [Prototype performance evaluation] [10 human-minutes + 30 compute-minutes + 1 GB disk]:

While experiments E2 and E3 are described separately in the paper, we bundle them together for convenience.

Preparation: If not yet complete, install the Intel SGX SDK driver and run the `setup.sh` script found in the

`scripts/` directory. The script installs all the necessary Intel SGX libraries and the ODT client and server. You can also use the provided Docker image where the `setup.sh` script is already complete. Also, make sure that you do not have any CPU intensive processes running while the measurements are performed.

Execution: To perform the measurements, run the `runtime_measurement.sh` script. This will configure and recompile the ODT client and server for runtime measurement. Afterwards, it will start various combinations of ODT and unmodified OpenSSL clients and servers, and perform six timing measurements in total with 1000 samples each. During each measurement, the script outputs the index of the current sample.

Results: To read the results, we provide the `analyze_results.py` script. It formats the results in the form of Table 2 and Table 3 from the paper to help evaluate claims (C2) and (C3). The raw results are stored in the `scripts/results/` directory.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: V-ORAM: A Versatile and Adaptive ORAM Framework with Service Transformation for Dynamic Workloads

Bo Zhang[†], Helei Cui[†*], Xingliang Yuan[◇], Zhiwen Yu[†‡], and Bin Guo[†]
[†]*School of Computer Science, Northwestern Polytechnical University, China*
[◇]*School of Computing and Information Systems, The University of Melbourne, Australia*
[‡]*College of Computer Science and Technology, Harbin Engineering University, China*
[*]*Corresponding author, email: chl@nwpu.edu.cn*

A Artifact Appendix

This artifact includes all implemented prototypes and the code/scripts for reproducing the major results/figures mentioned in the paper. We implement V-ORAM on-premise using Python's built-in libraries and the lightweight encryption library `pycryptodome`. Our source code, with a detailed README, is available at https://github.com/BoZhangCS/V-ORAM. The entire evaluation can be performed through simple command-line instructions. We also uploaded the artifact to Zenodo, the DOI is http://dx.doi.org/10.5281/zenodo.14732806.

A.1 Abstract

V-ORAM is a versatile and adaptive ORAM framework with service transformation for dynamic workloads. The provided artifact enables efficient transformations between Path ORAM, Ring ORAM, and ConcurORAM.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact is evaluated through publicly available anonymized datasets and local randomized read-write. It does not involve any security, privacy, or ethical concerns.

A.2.2 How to access

The source code of the artifact is available at https://github.com/BoZhangCS/V-ORAM. We also uploaded the artifact to Zenodo, the DOI is http://dx.doi.org/10.5281/zenodo.14732806.

A.2.3 Hardware dependencies

The artifact is evaluated locally without requiring hardware beyond a general-purpose CPU (no TEE required). During execution, the artifact persists ORAM data and maintains metadata (*e.g.*, PosMap). So please ensure the current directory has at least 90GB free storage and 16GB available RAM. Our test machine is a Mac mini equipped with 512GB storage and an M2 chip with 16GB RAM.

A.2.4 Software dependencies

The artifact has been adapted for Windows, macOS, and Linux. We evaluated it under a macOS Sequoia 15.3.2 system. It is implemented on `Python-3.8`, and we recommend using the same version. All required third-party libraries are listed in `requirements.txt` and can be installed directly via `pip`.

A.2.5 Benchmarks

The artifact only requires subsets of the datasets in the real-world case study, which are already included in the artifact at `data/real_workloads.zip`. The full version of three datasets are available through: MSRC: https://iotta.snia.org/traces/block-io/388, AliCloud: https://github.com/alibaba/block-traces, Twitter: https://github.com/twitter/cache-trace.

A.3 Set-up

After cloning the project, please follow the steps below to set up the artifacts (README.md#Installation).

A.3.1 Check for Storage

Please check whether the current directory has enough storage space by running the command "`df -h .`".

A.3.2 Set up the environments

Set up the virtual `Python` environment and install all third-party dependencies through the following commands:

```
pip3 install virtualenv
virtualenv .venv -python=python3.8
source .venv/bin/activate
pip3 install -r requirements.txt
```

A.3.3 Installation

Extract the stripped real-world datasets via commands:

```
cd ./data
unzip real_workloads.zip
```

A.3.4 Basic Test

Perform random read/write tests for prototypes:

```
pytest
```

A.4 Evaluation workflow

The artifact reproduces major plots, tables, and statistical data in the paper. We develop scripts for each experiment, which can be evaluated entirely through command line instructions. All experiments can be run in parallel, except Figures 7, 9, and 10, which require the results from Figure 5. Briefly, the evaluation procedures are as follows:

1. Get artifact from https://github.com/BoZhangCS/V-ORAM.

2. Set up the environment and install the necessary dependencies according to README.

3. Run the experiments through the command line instructions provided in README.

A.4.1 Major Claims

(C1): V-ORAM enables efficient ORAM service transitions. It incurs only constant-level transformation costs, which do not scale with the number of the buckets. This is proven by experiment (E1). It also introduces constant extra costs to ORAM services, as proven by experiment (E2). The storage cost of V-ORAM is $n \log n$ bits, as shown in experiment (E3).

(C2): The planner in V-ORAM assists clients in selecting appropriate ORAM parameters, based on the distribution of its datasets and request. This is proven by experiments (E4) and (E5).

(C3): V-ORAM is efficient and effectively reduces monetary costs in real-world scenarios. The planner helps the client decide whether to switch ORAM services. This is proven by experiments (E6) and (E7).

A.4.2 Experiments

(E1): [Figure 4] [1 human-minutes + 1 compute-minutes]: (E1) compares V-ORAM with the other two baselines (D and M) mentioned in §6.1 (README.md#Figure-4).
How to: All the preparations are stated in Artifact Appendix A.3. Execute the following command to reproduce our results of (E1):

```
python3 ./artifacts/Fig4_transformation.py -pdc
```

Results: The communication, computation, and RTT costs of three schemes under different ORAM sizes. Figures plotted from the results of the artifact. Comparison of the plotted figures and the figure in the paper.

(E2): [Figure 5] [1 human-minutes + 5 compute-minutes]: (E2) runs V-ORAM and records the costs of EvictRecord during the process. The three types of access cost in the figures are described in §6.2#"EvictRecord costs" (README.md#Figure-5).
How to: All the preparations are stated in Artifact Appendix A.3. Execute the following commands to reproduce our results of (E2):

```
python3 ./artifacts/Fig5_EvictRecord_prfm.py -pdc
```

Results: The costs of three types of V-ORAM access when running Ring ORAM or ConcurORAM services, under different ORAM sizes. Figures plotted from the results of the artifact. Comparison of the plotted figures and the figures in the paper.

(E3): [Figure 6] [1 human-minutes + 5 compute-minutes]: (E3) runs three ORAMs and record their stash size, and compares it with the sizes of PosMap and RecMap (README.md#Figure-6).
How to: All the preparations are stated in Artifact Appendix A.3. Execute the following commands to reproduce our results of (E3):

```
python3 ./artifacts/Fig6_storage_cost.py -pdc
```

Results: The storage costs and proportion of three ORAMs under different ORAM sizes. Figures plotted from the results of the artifact. Comparison of the plotted figures and the figures in the paper.

(E4): [Figure 7] [1 human-minutes + 1 compute-minutes]: (E4) runs three ORAMs under various system performance requirements and estimates their monetary costs under different bucket sizes (README.md#Figure-7).
How to: All the preparations are stated in Artifact Appendix A.3. Execute the following commands to reproduce our results of (E4):

```
python3 ./artifacts/Fig7_decision_choose_ORAM.py
-pdc
```

Results: The monetary costs of three ORAMs under different system performance requirements. Figures plotted from the results of the artifact. Comparison of the plotted figures and the figures in the paper.

(E5): [Figure 8] [1 human-minutes + 1 compute-minutes]: (E5) compares the communication blowup of three ORAMs under different parameters and block sizes (README.md#Figure-8).

How to: All the preparations are stated in Artifact Appendix A.3. Execute the following commands to reproduce our results of (E5):

```
python3 ./artifacts/Fig8_opt_block_size.py -pdc
```

Results: The communication blowup of three ORAMs under different parameter settings. Figures plotted from the results of the artifact. Comparison of the plotted figures and the figures in the paper.

(E6): [Figure 9] [1 human-minutes + 10 compute-minutes]: (E6) evaluates V-ORAM performance under two realistic settings (different RTT and block sizes). The RTT experiments are derived from the results of (E2), while the block sizes experiments require re-running V-ORAM (README.md#Figure-9).

How to: All the preparations are stated in Artifact Appendix A.3. Execute the following commands to reproduce our results of (E6):

```
python3 ./artifacts/Fig9_realistic_settings.py
-pdc
```

Results: The response time under different RTT and the processing time under different block size. Figures plotted from the results of the artifact. Comparison of the plotted figures and the figures in the paper.

(E7): [Figure 10] [1 human-minutes + 1 compute-minutes]: (E7) reads three real-world datasets and simulates V-ORAM according to its requests (README.md#Figure-10).

How to: All the preparations are stated in Artifact Appendix A.3. Execute the following commands to reproduce our results of (E7):

```
python3 ./artifacts/Fig10_real_world_case_studies.py
-d
```

Results: Established V-ORAM workloads for three datasets. Plotted figures from three datasets. Since the results are stable, there is no comparison.

(E8): [Figure 11] [1 human-minutes + 1 compute-minutes]: (E8) runs V-ORAM under our two simulated workloads and records performance during execution. These two workloads represent dynamic throughput and latency (README.md#Figure-11).

How to: All the preparations are stated in Artifact Appendix A.3. Execute the following commands to reproduce our results of (E8):

```
python3 ./artifacts/Fig11_simulated_workloads.py
-pdc
```

Results: The throughput of V-ORAM under two simulated workloads. Plotted figures from three datasets. Since the results are stable, there is no comparison.

(E9): [Table 3] [1 human-minutes + 1 compute-minutes]: (E9) reads the datasets, calculated statistics, and provides the estimate tree heights, recommended block sizes, and communication blowups under Ring ORAM (README.md#Table-3).

How to: All the preparations are stated in Artifact Appendix A.3. Execute the following commands to reproduce our results of (E9):

```
python3 ./artifacts/Tab3_real_case_study_para.py
-p
```

Results: The statistics and calculated results. Plotted figures from three datasets. Since the results are stable, there is no comparison.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: AUTOVR: Automated UI Exploration for Detecting Sensitive Data Flow Exposures in Virtual Reality Apps

John Y. Kim Chaoshun Zuo Yanjie Zhao Zhiqiang Lin
The Ohio State University

A Artifact Appendix

A.1 Abstract

This artifact is the source code of the tool: AutoVR. AutoVR's purpose is to instrument scenes, UI, and physics events on Unity VR apps ran on the Meta Quest 2. The source code of AUTOVR is available on GitHub: https://github.com/OSUSecLab/AutoVR, additionally, the experimental results of all 366 apps available at: https://doi.org/10.5281/zenodo.15832783. To evaluate AutoVR, we have developed a model test Unity VR app, which AUTOVR can comprehensively exercise all scenes and events in. Additionally, to show AutoVR's capabilities, we have instrumented AUTOVR upon the 366 VR apps and collected network traffic during instrumentation. The results of both experiments are compared with experiments using Android Monkey, a black-box Android event testing tool, in-place of AutoVR.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Please ensure account and device usage permission has been granted by the owner of the Meta Quest 2 device as VR apps are known to collect user data and could pose a risk to the owner's privacy. Our dataset uses test accounts and test devices, and no PII data were collected from humans.

A.2.2 How to access

AUTOVR and our datasets are available at https://doi.org/10.5281/zenodo.15832783, in addition to 10 real-world Unity VR APKs to run AUTOVR with. As AUTOVR continues to be developed, we have made AUTOVR available on GitHub at https://github.com/OSUSecLab/AutoVR.

A.2.3 Hardware dependencies

AUTOVR can be run on any standard machine running Linux or MacOS. A Meta Quest 2 device with developer mode (i.e., capable of using `adb`) is required to run AutoVR. While AUTOVR does not require a rooted Meta Quest 2 device, it is recommended to use a rooted device as a significant subset of the 366 apps were run on a rooted device.

A.2.4 Software dependencies

Software dependencies for AUTOVR are detailed in the README file on AutoVR's GitHub page.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

We have provided the complete instructions on how to install AUTOVR on the README page on GitHub. For the experiments, we include two APK files in the Zenodo repository: *model-app-non-rooted.apk*, and *antmonitor-modified.apk*. We describe the purpose for each APK in the following:

- *model-app-non-rooted.apk* is the model app embedded with `frida-gadget` used in §6.1 in the evaluation of the paper.
- *antmonitor-modified.apk* is a modified version of *AntMonitor* suited for the sensitive data exposure detection experiment, detailed in section §6.2 of the paper.

To install an APK file, run the following command: `adb install <app>.apk`

A.3.2 Basic Test

Once AUTOVR and the model app ((a) or (b) from A.3.1) are installed, follow the steps listed in the README file to build and run AutoVR. Use the following configurations from the Flag Table:

- *device*: (ID of the Meta Quest 2 using `adb devices`).
- *package*: *com.osuseclab.AutoVRTest*

If the device is rooted, ensure the `-rooted` flag is set. To test functionality, the expected output should show a log containing "Starting AutoVRApp", and initialization logs showing a list of (.dll) assembly names. AUTOVR should then be exercising events on the model app until the finished log shows. The effects of the events will be reflected visually in the Meta Quest 2 device.

A.4 Evaluation workflow

A.4.1 Major Claims

Our paper presents AutoVR, a tool/framework to automatically insturment scenes, UI, and physics events on Unity VR apps. In the paper, we claim that AUTOVR can indeed exercise events by leveraging the internal binary, and be used to collect any sensitive data exposures from the app using network collection tools like *AntMonitor*. We make the following major claims:

(C1): By leveraging the internal binary, AUTOVR is capable of exercising scenes, UI, and physics events from a model Unity VR test app. This is proven by experiment (E1) whose results are shown in Table 2 in section §6.2 of the paper.

(C2): AUTOVR can be used to exercise functions that expose sensitive data from outgoing traffic in a Unity VR app. This is proven by experiment (E2) whose results are shown in Figure 9 and Figure 10 in §6.2 of the paper.

(C3): AUTOVR significantly outperforms Android Monkey in both exercising events and exposing sensitive data from a Unity VR app. This is proven by both experiments (E1) and (E2), whose results are shown in Figure 11 in section §6.2 of the paper.

A.4.2 Experiments

(E1): *[Event exercising capability using the model test Unity VR app] [5 human-minutes + 200 compute-seconds + 5MB disk]: This experiment will test* AUTOVR *and Android Monkey on a model test app, where the number of scenes and events are known. When an event executes, the test app will log the number of executions and output each scene's events in locally stored json files.*

Preparation: Ensure that a Meta Quest 2 device is available with developer mode enabled (i.e., `adb` commands can be performed for the device). Perform the installation of AUTOVR and the *model-app(-non-rooted).apk* app on the Meta Quest 2 device described in the A.3.1. Ensure no other apps are running on the device.

Execution: Run AUTOVR with the instructions listed in the README file on AUTOVR's GitHub page, with the *package* argument set to *com.osuseclab.AutoVRTest*. Note: sometimes AUTOVR can initially stall. If the initialization of the assembly (.dll) files are not shown in the output log after 5 seconds, cancel execution and rerun AutoVR.

Results: Collect the results from pulling all JSON files (.json) from the following directory within the Meta Quest device: `/sdcard/Android/data/com.osuseclab.AutoVRTest/files/`. Each JSON file is named with the following scheme: <*event-type*>-<*scene name*>*.json*, where the <event-type> indicates what type of event (i.e., UI, trigger, collision) and <scene name> being the name of the scene. The JSON file structure is keyed by the event name and valued by the number of times executed. Each physics event key is schemed in the following format: <*collider name*>|<*physics event type*>|<*the other collider to perform the event*>. The outputted results from using AUTOVR and Android Monkey should reflect Table 2 in the paper.

(E2): *[Sensitive data exposure capability using real Unity VR apps] [2 human-hour + 3 compute-hour]: This experiment will test* AUTOVR *and Android Monkey on 5 real-world Unity VR apps to compare the number of data exposures exercised by* AUTOVR *and Android Monkey. Testing all 366 Unity VR apps will take weeks of instrumentation, as such, we have scaled down the number of apps to 5 for reproducibility.*

Preparation: Install all 5 Unity VR APKs (available at https://doi.org/10.5281/zenodo.15832783) onto the Meta Quest 2 device. Additionally, install the *antmonitor-modified.apk* file from the Zenodo repository. Pull and unzip the Monkey source code: `monkey.zip` file from the Zenodo repository. Ensure no other apps are running on the device for each time you run AUTOVR or Monkey.

Execution: We have provided the script `run_with_antmonitor.sh` for both AutoVR and Monkey. `run_with_antmonitor.sh` requires three arguments:

- *device_name*: The Meta Quest 2 device name from `adb devices`.
- *package_name*: The package name of the app.
- *ssl_offset*: The SSL offset for SSL unpinning.

The SSL offsets for all 5 apps have been precomputed and are available at https://doi.org/10.5281/zenodo.15832783. For each package name listed, use the SSL offset for the *ssl_offset* argument. The following is an example command for one of the test apps: `./run_with_antmonitor.sh FakeDeviceName com.BMINC.ArtGateVR 0x76efa8`

Results: After each run of AUTOVR and Android Monkey, there will be a subdirectory called `results` which contain the network traffic triggered by the tools. The expected results should show that the AUTOVR runs extracted more bytes from the PCAP files for each package name than the Monkey runs.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: *Found in Translation*: A Generative Language Modeling Approach to Memory Access Pattern Attacks

Grace Jia
Yale University

Alex Wong
Yale University

Anurag Khandelwal
Yale University

A Artifact Appendix

A.1 Abstract

Confidential computing environments (CCEs) offer a convenient way for privacy-sensitive applications to ensure confidentiality and integrity for data and computations offloaded to the cloud. However, the Operating System (OS) stack is still controlled by the cloud provider and manages key system services such as memory paging. Several recent works have demonstrated that these services can leverage the side channel of page access patterns to reconstruct the application's private data. However, prior attacks are limited to identifying accesses to a specific application-level object by a unique access or sequence of accesses to OS-level pages. Moreover, they tend to ignore correlations in access patterns — a common occurrence in most real-world applications — leaving untapped critical side channel information for improving attack accuracy.

We propose a novel attack approach, named *Found in Translation* (FIT), that leverages access correlations across pages in cloud applications using a generative language model. Our key insight is that there are strong parallels between application page access patterns and grammatical structures in natural languages, making language modeling an excellent fit for reconstructing sensitive application data with high accuracy. Our major results are the high accuracy of the attack compared to prior state-of-the-art approaches in predicting object-level access sequences on three privacy-sensitive applications: DLRM inference for recommendations, LLM inference for medical diagnosis, and HNSW index lookup for semantic search. We also include results on our attack's robustness against errors in the recorded page access sequences and the latency that OS-level tracking and recording page accesses add to the execution times of the victim applications.

A.2 Description & Requirements

This artifact contains the following: (i) the source code to train and evaluate the language model used in FIT, along with code to run the compared baselines — the IHOP attack and the Naive Bayes classifier; (ii) the preprocessed datasets used in our evaluation to train and test all compared attacks, FIT's trained model weights for each evaluated application, and our measurements of application execution times with and without page access tracking; (iii) the scripts required to reproduce our experiments and a comprehensive README with instructions to set up and run the FIT attack pipeline.

A.2.1 Security, privacy, and ethical concerns

This artifact does not take destructive steps, nor does it disable security mechanisms. It does not pose any privacy or ethical concerns to evaluators.

A.2.2 How to access

Our artifact is available on Zenodo and can be accessed at https://doi.org/10.5281/zenodo.15602651.

A.2.3 Hardware dependencies

We evaluated our FIT attack using a single NVIDIA GeForce RTX 4090 GPU; however, we expect FIT to run on any NVIDIA GPU with at least 16 GB of memory. We also provide an option for CPU-only platforms to run our experiments on a smaller subset of test samples.

The compared attacks, IHOP and Naive Bayes, require only CPU hardware to evaluate. We have previously run their experiments on AMD EPYC 7302P 16-core processor, as well as Apple M2 Pro 12-core processor with 16 GB RAM.

A.2.4 Software dependencies

We conducted experiments involving FIT on Ubuntu 24.04.1 LTS. Experiments involving compared attacks have been conducted on Ubuntu 22.04.2 LTS and macOS Sonoma 14.6.1. All attacks require Python3, and the plotting script requires LaTeX to be installed. We use Conda for environment management and document all required packages in the file 'requirements.txt' in the root directory of the artifact.

A.2.5 Benchmarks

We evaluate our compared attacks on page access traces collected from three applications: a Deep Learning Recommendation Model (DLRM), a Large Language Model (LLM), and a Hierarchical Navigable Small World index (HNSW); more details about the datasets can be found in our paper. All required data and model weights are provided on Zenodo.

A.3 Set-up

We assume that the system has at least 22 GB of available disk: 17 GB for the artifact, and 5 GB for the Conda environment.

A.3.1 Installation

We recommend using Conda to set up our experimental environment. If Conda is not installed on the system, you can quickly set up Miniconda following these instructions. You may need `wget` or `curl` to download the installer.

First, create a new Conda environment with Python 3.12:

```
conda create -n fit -y python=3.12
conda activate fit
```

Next, install the required dependencies:

```
pip install -r requirements.txt
```

A.3.2 Basic Test

The following script runs FIT, IHOP, and the Naive Bayes baseline on a single test sample from our LLM dataset:

```
bash scripts/run_basic_test.sh
```

We expect the tests to take around 5 minutes in total. Each test populates the 'data/llm/eval' directory with a results file — respectively, 'llm_nitro.csv', 'nb_llm.csv', and 'ihop_llm.pkl'. Before running our experiments, please remove these files with `rm data/llm/eval/*`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): FIT achieves significantly lower hamming distances (i.e., more accurate predictions) than the compared Naive Bayes and IHOP baselines across all three use cases, DLRM, LLM, and HNSW. This is demonstrated by experiments (E1), (E2), and optionally (E3), the results of which are shown in Figure 7.

(C2): Attack efficacy improves for all three attacks given a one-to-one mapping of OS-level pages and application-level objects, but FIT is still more accurate than IHOP. This is also demonstrated by experiments (E1), (E2), and (E3), and the relevant results are shown in Figure 8.

(C3): FIT is robust against measurement errors in the input page access traces. This is demonstrated by experiment (E4), the results of which are shown in Figure 9.

A.4.2 Experiments

Preparation: Before running each script, set the current directory to the root directory of the artifact and activate the `fit` Conda environment.

Results: If successful, each experiment will save its results in directories corresponding to each use case: 'data/dlrm/eval/', 'data/llm/eval/', and 'data/hnsw/eval/'.

(E1): *[FIT Attack Efficacy] [3.5 GPU compute-hours]:* Runs the inference phase of FIT on test samples from the three use cases for (C1) and the one-to-one use case for (C2).

Execution: To run the experiment with the paper's test dataset sizes for DLRM, LLM, and HNSW:

```
bash scripts/run_fit.sh \
100000 50000 2600
```

If only CPU is available, we recommend running the DLRM and LLM experiments with fewer test samples:

```
bash scripts/run_fit.sh \
1000 5000 2600 --use-cpu
```

Results: The results of the FIT experiment are saved as the following files in the result directories corresponding to their use case: 'dlrm_nitro.csv', 'dlrm_sgx.csv', 'llm_nitro.csv', 'llm_sgx.csv', 'hnsw_nitro.csv', 'hnsw_sgx.csv' for (C1), and finally 'dlrm_1_1.csv' for (C2). Each row of the CSV contains an access sequence predicted by the FIT model and the corresponding ground-truth target sequence. After collecting the results of (E2), the following script can reproduce Figures 7 and 8 from our paper and save them to the 'plots' directory:

```
python3 scripts/plot.py --fig 7
python3 scripts/plot.py --fig 8
```

(E2): *[Naive Bayes Attack Efficacy] [1.5 CPU compute-hours]:* Runs the Naive Bayes baseline on the train and test data of the three use cases for (C1) and the one-to-one use case for (C2).

Execution: `bash scripts/run_nb.sh`

Results: Like (E1), results of the Naive Bayes runs are saved in the following CSV files: 'nb_dlrm.csv', 'nb_llm.csv', 'nb_hnsw.csv', and 'nb_dlrm_1_1.csv'. Each row of each CSV is a tuple of the predicted and target sequences.

(E3): *[IHOP Attack Efficacy] [56 CPU compute-hours]:* Runs the IHOP attack on the train and test data of the three use cases (C1) and the one-to-one use case (C2). The compute-hours are estimated based on prior runs with the AMD setup, but we have observed an up to 2x speedup on the Apple M2 Pro.

Execution: `bash scripts/run_ihop.sh`

Results: Each run of IHOP outputs a list of predicted and target sequences, which are saved as

the following pickle files in the result directories: 'ihop_dlrm.pkl', 'ihop_llm.pkl', 'ihop_hnsw.pkl', and 'ihop_dlrm_1_1.pkl'. As this experiment is relatively time-intensive, our artifact provides results from our runs of IHOP in the parent directories 'data/dlrm/', 'data/llm/', and 'data/hnsw/'. These provided results are automatically used by the plotting script if there is insufficient time to run (E3).

(E4): *[*FIT *Practical Considerations] [6.5 GPU compute-hours]:* Runs the inference phase of FIT on test samples from the three use cases with varying measurement error rates: 1%, 3%, 5%, 7%, and 10% (C3).

Execution: To run the experiment:

```
bash scripts/run_fit_sensitivity.sh
```

Like (E1), CPU-only options are available:

```
bash scripts/run_fit_sensitivity.sh \
1000 5000 2600 --use-cpu
```

Results: The results of each run of FIT inference are saved as a CSV, with each row containing the access sequence predicted by the FIT model and the corresponding ground-truth target sequence. The following script can then recreate Figures 9 and 10:

```
python3 scripts/plot.py --fig 9
python3 scripts/plot.py --fig 10
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Current Affairs: A Security Measurement Study of CCS EV Charging Deployments

Marcell Szakály
University of Oxford

Sebastian Köhler
University of Oxford

Ivan Martinovic
University of Oxford

A Artifact Appendix

A.1 Abstract

Our paper investigates via measurement study the security of DC CCS EV charging stations across Europe. Our artifacts are divided into three parts: The data collection system consists of specialized hardware that connect to a CCS charging station, and software to emulate a vehicle and collect data about charging sessions. Our dataset contains both machine and human readable information about the devices we tested. Finally, we provide the data analysis code used to view and extract statistics from the dataset.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The code of our artifacts is not expected to harm any host system, it reads and writes to files inside of artifact related directories.

Building and using our data collection hardware must be done with care, to ensure that unsafe voltages coming from EV chargers are appropriately contained and that the device does not harm chargers. Our system is designed to never enable the high voltage output of EV chargers, and it does not require connecting to any of the high voltage pins of the connector. However, for practical reasons, using the data collection system is not part of this artifact evaluation.

A.2.2 How to access

The evaluated version of our artifacts can be found permanently on Zenodo at https://zenodo.org/records/14712106. The code repositories are also available on GitHub, the data collection system at https://github.com/ssloxford/current-affairs/, and the data analysis system at https://github.com/ssloxford/current-affairs-analysis.

A.2.3 Hardware dependencies

Our dataset and analysis code use basic portable Python features, and require minimal resources (<1GB disk, <1GB RAM).

The data collection system requires specific hardware: A Raspberry Pi, a Homeplug Green PHY modem, a custom PCB for electrical signalling, and a CCS inlet. In addition, to collect data, the device must be connected to a DC CCS EV charger. It is possible to activate test mode, which removes requirements for any special hardware, but in this mode no useful experiments can be performed.

It is not possible for us to provide remote access to this hardware throughout the artifact evaluation period, since for any meaningful results it would have to be connected to an EV charger and require interacting with buttons on the charger.

A.2.4 Software dependencies

The data analysis was tested using Python 3.9 on Windows 10 and Python 3.12.3 on Kubuntu 24.04.1 LTS, but we expect it to work on other mainstream modern Linux and Windows versions. It requires Python, Jupyter, and common Python packages.

The data collection system runs on Raspbian, and requires low level hardware access. The installation process requires changing core system settings and requires root access, but the application itself does not require root. It requires Python, Java, Maven, Git, and a C compiler. The main code is written in Python 3.9, and relies on standard packages. In addition, it requires a compiled C Python module, the source code and build instructions are provided for this. Finally, a modified version of the V2Gdecoder https://github.com/FlUxIuS/V2Gdecoder application is needed to convert binary EXI messages used by CCS into XML. For this we provide a "diff" file with our modifications. The application further requires XML schema files for the protocols, which are available (after purchase) as part of the DIN 70121, ISO 15118-2 and ISO 15118-20 standards. We provide instructions on where to place these files once obtained. As part of these AE process, it is possible to start the data collection system without any of these schema files, as they are only required in later stages of the CCS protocol.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Dataset For the dataset alone, download and extract `data.zip` from Zenodo. The setup script for the data analysis (see below) automatically downloads and places this in the `data` subfolder of the analysis repository.

Data analysis For the data analysis, our instructions here assume x64 Kubuntu 24.04.1 LTS and Python 3.12, see the readme file for more general instructions. The ISO image to install the operating system can be found at https://cdimage.ubuntu.com/kubuntu/releases/24.04.1/release/, and it can be used in a Virtual Machine. First, download the files: the artifact evaluated version is available on Zenodo, and will be updated with the final version.

```
# wget https://zenodo.org/records/15119591/files/analysis.zip?download=1 -O analysis.zip
# unzip analysis.zip -d analysis
# cd analysis
```

We provide an install script to download and install all dependencies. It requires root in order to run `apt install` commands.

```
# chmod +x install.sh
# ./install.sh
```

All Python dependencies are installed in a virtual environment, this command is necessary to activate it in any new shell instance.

```
# source venv/bin/activate
```

Data collector For the data collection system, we do not expect reviewers to have the necessary hardware and time to create the full system. Instructions for doing this are described in the readme file, while here we outline minimal steps to verify compilation and partial execution, using no special hardware, Kubuntu 24.04.1 LTS and Python 3.12. First, download the files. The latest version is available on GitHub, and Zenodo will be updated with the final version.

```
# wget https://zenodo.org/records/15119591/files/collector.zip?download=1 -O collector.zip
# unzip collector.zip -d collector
# cd collector
```

We provide an install script to download and install all dependencies. It requires root in order to run `apt install` commands.

```
# chmod +x install.sh
# ./install.sh
```

This downloads the necessary Java app, C module code, and compiles them. All Python dependencies are installed in a virtual environment, this command is necessary to activate it in any new shell instance:

```
# source venv/bin/activate
```

If running without the real hardware, it is important to disable hardware access. To do this, edit `code/utils/settings.py`, and change:

```
SKIP_BASIC = True
SKIP_SLAC = True
```

Further, edit `code/main_ev.py`, and replace `"eth0"` on line 37 with the name of an Ethernet interface to bind to (you can obtain a list from `# ifconfig`)

A.3.2 Basic Test

Dataset To test the dataset, in the data folder open `index.html` and browse the data.

Data analysis Make sure the virtual environment has been activated. To test the analysis system, in the root of the analysis folder run the file `plots.ipynb`, which should output the plots found in the paper into the `plots` folder. To run it as an application, and output plots into the `plots` folder, use:

```
# python3 -m jupyter execute plots.ipynb
```

Data collector The install script should succeed without error, indicating that everything has been compiled and downloaded successfully. Make sure the virtual environment has been activated. Start the data collector by running:

```
# python3 main_harness.py
```

and open https://localhost:8000. The server uses a self signed certificate, which most browsers prompt the user to accept. If the UI prompts "Click here to activate WS certificate" follow this link and accept the self signed certificate a second time, then return to the original page. The UI should now show the "Start" button, this confirms that the webserver is working as expected.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We publish detailed information about our dataset, including location, serial number and photos of each charger (where known) as well as the test results for each. This is validated by E1.

(C2): Our dataset contains 325 chargers, with the composition described in Section 3.3 of the paper. These values were derived from our dataset using E2.

(C3): Our data indicates that many CCS deployments do not use TLS and use old standard versions, with exact numeric results throughout Section 4. These values were derived from our dataset using E2.

A.4.2 Experiments

(E1): *[30 human-minutes + negligible compute resources]: Validate availability of dataset*

Preparation: Install the dataset.

Execution: Open `index.html` in the dataset folder with a browser. Click onto various charger entries and look at them.

Results: Confirm that our dataset contains information about tested chargers, and results.

(E2): *[30 human-minutes + negligible compute resources]: Validate results derived from the dataset*

Preparation: Install the data analysis system following the instructions above (or using the readme file in the analysis repo).

Execution: Activate the virtual environment of the analysis repo and open the notebook:

```
# python3 -m jupyter notebook plots.ipynb
```

Run all cells in the notebook.

Results: The notebook should run without error. The first few cells of the notebook calculate the statistics claimed in Section 3.3. The next few cells calculate the results claimed in 4.2, 4.1 and 4.3, as well as generate the plots. Table 2 is saved to `output/all_data.tex`

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: "STEK Sharing is Not Caring: Bypassing TLS Authentication in Web Servers using Session Tickets"

Sven Hebrok
Paderborn University

Tim Leonhard Storm
Paderborn University

Felix Matthias Cramer
Paderborn University

Maximilian Radoy
Paderborn University

Juraj Somorovsky
Paderborn University

August 6, 2025

A Artifact Appendix

A.1 Abstract

Our artifact consists of three separate software submissions mapping to the three major parts of our paper:

Artifact I A tool for offline scans evaluating the impact of different SNI and `Host` header combinations on server contents when using session resumption.

Artifact II A tool for offline scans evaluating the impact of different SNI and `Host` header combinations on TLS client authentication when using session resumption.

Artifact III A tool(chain) for online scans evaluating the impact on served content when using session tickets issued by related, but different hosts.

Artifact I and Artifact II can be used to fully reproduce our results mostly offline. Both only require internet access to download docker containers. Artifact III can be used to reproduce the *online*, but due to servers changing (partially in response to our disclosure), the results will likely differ. We provide a slimmed down version running against local Docker containers instead, which demonstrates the workflow and functionality of our software.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Artifacts I and II download and run Docker images from the official registry. All containers use the official images from the respective web server maintainers. We also create our own Docker containers based on the official Docker-in-Docker images to run our artifacts. These containers are run with the `--privileged` flag.

A.2.2 How to access

The artifacts are made permanently accessible on zenodo, available at

`https://doi.org/10.5281/zenodo.15474656`

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Because all three artifacts use radically different setups, we provide Docker images for Artifact I and Artifact II. Extended instructions are contained in the respective README.

Artifact I OpenSSL, Docker

Artifact II OpenSSL, Docker

Artifact III Python $\geq$ 3.12, Pip, Docker, Golang 1.20, cmake, jqo, libjudy, libgmp, libpcap, byacc, flex, libjson, libunistring, curl, git

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The instructions shown here use the dockerized versions where possible, but instructions for running on baremetal are available in the README.

Artifact I Build the Docker container:

```
cd server-tests
docker build -t server-tests:latest .
```

Artifact II Build the Docker container:

```
cd Auth_Dotsche
docker build -t authdotsche:latest . -f
↪ Dockerfile
```

Artifact III *We strongly recommend using a VM, because we assume sudo for a lot of operations.* Build all required tools (including our respective forks):

```
git clone --recurse-submodules ...
sudo ./build_dependencies.sh
```

A.3.2 Basic Test

Artifact I Spawn a pre-configured `nginx` Docker container using

```
docker run --rm --name server-tests -it
↪ --privileged server-tests:latest deploy
↪ nginx one-server
```

`curl` commands for different SNIs are printed, that should result in different dummy websites being received. Execute these via

```
docker exec server-tests <curl cmd>
```

Artifact II Omitted because the complete test run is fully automated and should only take minutes on modern mid-range hardware. No reasonable subset of setups or checks would simplify the setup or shorten the runtime for a considerable amount.

Artifact III Check that the different binaries are built correctly:

```
./zdns/zdns --help
./zmap/src/zmap --help
./zmapv6/src/zmap --help
./zgrab2_tls13/cmd/zgrab2/zgrab2 --help
```

Also check whether the dummy servers and DNS resolution are working:

```
./setup.sh
dig @127.0.0.1 -p 8053 a.com
```

This should yield some local Docker IP.

A.4 Evaluation workflow

Because of the differences between our artifacts, we split this section into three parts.

A.4.1 Major Claims Artifact I

(C1): We claim that Table 1 presented in Section 4 is accurate for the versions listed in appendix A.

A.4.2 Experiments Artifact I

(E1): Generating software results *[2 human-minutes + 15 compute-minutes]*:
Preparation: Install the software according to appendix A.3.1.
Execution: Run to execute all testcases against all supported software configurations.

```
docker run --privileged -it -v ./out:/out
↪ server-tests:latest evaluate --outdir
↪ out
```

Optionally, you can restrict the output to a single software or testcase with `--software` and `--case` respectively. A full list of available options can be found in `testcases/configuration.yml` By default results are written in the working directory, but `--outdir` can be used to change the directory (Note that this refers to the directory inside the container for dockerized usage).
Results: The file results.{json, csv, jsonl} lists how each of the different softwares and their configurations react to different SNI and `Host` header, based on the virtual host issuing the session ticket and the receiving virtual host. This is the raw data for Table 1. Note that the table contains only the `one-server` case.
You can either verify the results against Table 1 by hand or refer to **(E2)**.

(E2): Verifying the table *[5 human-minutes + 1 compute-minute]*:
Preparation: Run (E1) in **full** configuration (without `--software` or `--case`).
Execution: Run

```
docker run --privileged -it -v ./out:/out
↪ server-tests:latest postprocess
↪ out/results.jsonl
```

and verify that the final two lines read "Validated table assumptions for ...".
Results: The postprocess script has different assertions about the results based on each software, e.g. checking that real behavior matches documented behavior or that different configurations behave the same. The output will list all these assertions, color-coded on whether they are as documented or whether they diverge.
Then the data is specifically checked against the behavior presented in Table 1. The corresponding code can be found in `evaluate/functionality/postprocess.py:394f` and `450f` in the `_check_table_assumptions...` functions.
Within each function, assertions are grouped by columns in the results table then rows. For example, the assertions in lines 396/397 and 454-460 check the results for the `SNI=I,H=I` case.
*Running this successfully explicitly confirms claim **(C1)**.*

A.4.3 Major Claims Artifact II

(C2): We claim that Table 2 presented in Section 4 is accurate for the versions listed in appendix A.

A.4.4 Experiments Artifact II

(E3): Verifying the Table *[10 human-minutes + 10 compute-minutes + 2GB disk]*

Preparation: Install the software according to appendix A.3.1.

Execution: Start the container. This will automatically run all tests against all web server configurations.

```
docker run --rm -v ./out:/code/out
↪ --privileged --name authdotsche
↪ authdotsche:latest
```

Alternatively, the provided file `run.sh` fully automates both steps, including the preparation.

Results: The experiment outputs its results both as console output and also as an HTML table. The resulting HTML table contains the full list of test cases and test results, which corresponds to the results presented in Table 2. The table will be saved in the mounted directory `./out/`.

You can verify the results by hand: Each column `A -> BC` describes the test case, where a ticket obtained with SNI and `Host` header A is then used with SNI B and `Host` C[1]. The first column `A->AA` (Ticket from Host A to SNI=A,Host=A) maps to the left most table column and so on.

Each row describes a different configuration: For example,

```
closedlitespeed_subdomains_certa_defaulta
    _strict_default
```

uses LiteSpeed, the two virtual hosts are on subdomains, the initial host[2] requires a certificate, is set as the default host and finally the software-specific `strict` setting is enabled.

The generated `results.html` includes a search field at the bottom, which filter the rows by regex, to improve usability. If you search `apache` you get all the `apache` variants, but you will not have to check all of them separately, as cells will be annotated with a colored tooltip, when behavior differs between variants (i.e. TLS 1.2 vs 1.3, A with client authentication vs freely accessible A, domain vs subdomain). For example, `apache A->nA` (so the SNI=none, Host=I column) gets marked because there is diverging behavior based on which host is set as a default, which matches the cross annotation in Table 2 at that entry.

*Verifying the table confirms claim **(C2)**.*

[1] n = not given, X = unavailable domain

[2] **I** in paper, **A** in code

A.4.5 Major Claims Artifact III

(C3): We claim that our software scans pairs of virtual hosts with the same STEK, following the strategy outlined in Section 5.1.

(C4): We claim that, combined with human effort, this was used to retrieve the results presented in Section 5.2 and can be used to run the study again.

(C5): We claim that the suggested countermeasures in Section 7 are sensible.

A.4.6 Experiments Artifact III

(E4): Run local scan *[2 human-minutes + 5 compute-minutes]*:

Preparation: Run `./setup.sh` in background.

Execution: Run `./0_all_in_one.sh`

Results: The script will invoke our large-scale scanning pipeline.[3] It demonstrates all steps of the process, including initial resolution and ticket prefixes, importing a graph DB for querying scan candidates, performing the actual scan by using session tickets cross hosts and subsequent analysis. The relevant output is the resulting folder `analysisdump` (cf. **E2**). Note that the code also contains automated classification (`UNSAFE`), but all actual results stem from the manual part (`LOOK_AT_METRICS`).

*Running this successfully addresses claim **(C3)**.*

(E5): Examining the local results *[15 human-minutes]*:

Preparation: Run **(E4)**.

Execution: Manually examine the contents of the `analysisdump` folder.

Results: The folder contains a list of all potentially successful resumptions across hosts, grouped by ASes. Within, each path `./<target IP>/<source IP>/` contains a separate scan result. I.e., the folder `./172.19.0.5/172.19.0.3/` contains the different resulting HTML documents when resuming at `.5` with a ticket from `.3`: `0_initial.html` is the original page at `.3`, without a ticket. `1_resumed.html` contains the page received from `.5` after resumption. `2_*_supposed_origin.html` contains the closest HTML match, for `1_resumed.html`, based on different metrics. `_meta.md` summarizes these findings, including which domain we believe to have encountered.

We then manually examined these cases to find the vulnerable hosts. *Comprehending this workflow addresses claim **(C4)**.*

(E6): Retry with different countermeasures *[see above]*:

Preparation: Enable one of the countermeasures by editing `ae-dummy-servers/docker-compose.yml` in line 12f.

Execution: Re-run experiments **(E4)**,**(E5)**.

[3] Changes have been made for running locally, but are marked with comments "AE Version"

Results: Running the previous experiments with different countermeasures enabled will yield different results (or none at all), indicating that fewer or no potentially unsafe resumptions were possible in the first place. *This supports claim **(C5).***

A.5 Notes on Reusability

Artifact III Our large-scale scanning pipeline is quite fragile at large sample sizes and we have experienced memory leaks, which may require occasional restarts. We also used Neo4J community version, leading to a workaround, as it only allows one database per container. Note, that the generated data may be upwards of 1TB, although our scripts have options for filtering out certain fields from zgrab etc. Nonetheless, the full pipeline can be easily restored from our submitted version to re-run the experiments if desired.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Unsafe LLM-Based Search: Quantitative Analysis and Mitigation of Safety Risks in AI Web Search

Zeren Luo[1]* Zifan Peng[1]* Yule Liu[1] Zhen Sun[1] Mingchen Li[1,2] Jingyi Zheng[1] Xinlei He[1]†

[1]*The Hong Kong University of Science and Technology (Guangzhou)* [2]*University of North Texas*

A Artifact Appendix

A.1 Abstract

This artifact encompasses the full implementation of our risk-mitigation agent as detailed in "Unsafe LLM-Based Search: Quantitative Analysis and Mitigation of Safety Risks in AI Web Search". It includes a prompt templates for content refinement and malicious URL detection, the Python code for constructing and operating the agent with options for an XGBoost-Detector, PhishLLM-Detector, and our HtmlLLM-Detector. Evaluators can utilize this artifact to refine AIPSEs' output content using our designed chain-of-thought prompt and classify URLs as benign or malicious using the above three detectors. All code is provided, and executing the basic tests confirms the proper functionality of each component.

A.2 Description & Requirements

This section lists all the information necessary to recreate the experimental setup used to run our artifact. It covers minimal hardware and software requirements, as well as access to our artifact.

A.2.1 Security, Privacy, and Ethical Concerns

The artifact does not perform any destructive operations or disable any security mechanisms on the evaluator's machine. It does make outbound HTTP(s) requests in two ways:

- **LLM Calls to OpenAI.** The `is_malicious` function and the content-refinement tool both send content or HTML snippets to OpenAI's API. Evaluators must supply a valid `OPENAI_API_KEY`, and they should be aware that all content sent to the LLM may be logged by OpenAI. No personal or sensitive data beyond URLs should be included.
- **URL Fetches.** When using the HtmlLLM-Detector variant (URL Detector 4), our code fetches remote pages either via `requests` or via Selenium. Evaluators should be mindful that fetching arbitrary URLs can expose them to potentially malicious or untrusted web pages. It is recommended to run URL-fetching in an isolated environment (e.g., a VM or container) if there is any concern about downloading unknown content.

No private or human-identifiable information is requested at any point. The code does not store or transmit any personal user data beyond what is strictly necessary for detecting phishing, malware, or scam content (i.e., the URLs and HTML code themselves). There are no ethical issues with running the provided code, provided the evaluator does not deliberately experiment on known malicious sites in an uncontrolled environment without caution.

A.2.2 How to Access

We share our artifact via a DOI-archived link in Zenodo:

DOI: `10.5281/zenodo.15607878`

URL: `https://doi.org/10.5281/zenodo.15607878`

The artifact includes all the defense code for our paper and a `readme.md` file that details the structure and usage of our defense method. We highly recommend verifying our code through the `readme.md` file.

A.2.3 Hardware Dependencies

The artifact can be evaluated on any standard x86_64 (or ARM64) machine running a recent Linux, macOS, or Windows OS. No GPU is required unless the evaluator wishes to accelerate the XGBoost model training for further experimentation (but all pre-trained weights are provided). Selenium-based crawling requires that a compatible Chrome/Chromium version and the corresponding ChromeDriver are installed. In summary:

- CPU: 64-bit processor (Intel/AMD/Apple Silicon)
- RAM: $\geq$ 4GB
- Disk: $\geq$ 5GB free for code and dependencies

*Contributed equally and listed alphabetically.
†Corresponding author (xinleihe@hkust-gz.edu.cn).

A.2.4 Software Dependencies

Operating System:
Linux (Ubuntu 20.04+), macOS (10.15+), or Windows 10/11.
Language and Packages:

- `Python 3.10+`
- `pip` (Python Package Installer)
- Required Python Packages:
  - `openai`
  - `langchain`
  - `langchain-openai`
  - `bs4`
  - `python-whois`
  - `pandas`
  - `selenium`
  - `requests`
  - `xgboost`

A.2.5 Benchmarks

None, we are sharing only the defense code, as we mentioned in the open science section.

A.3 Set-up

A.3.1 Installation

1. **Create a Python Virtual Environment (Recommended):**

```
conda create -n testenv python=3.10
conda activate testenv
```

2. **Install All Python Dependencies:**

```
pip install --upgrade pip
pip install -r requirements.txt
```

3. **(Optional) Install ChromeDriver for Selenium** Download the ChromeDriver at `https://developer.chrome.com/docs/chromedriver/downloads`
4. **Set Environment Variables:** Please see the `readme.md` file, without a valid `OPENAI_API_KEY`, the LLM-based detectors and the content-refinement tool will exit with an error.

A.3.2 Basic Test

Please see the `readme.md` file to run the basic test, as the content is not displayed clearly in LaTeX due to formatting issues.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): HtmlLLM-Detector achieves the highest F1 score among the three detectors. This is demonstrated by the experiment (E1) described in Section 8.2 of the paper, with results illustrated in Table 2 and Table 3 of the paper.
(C2): Agent-based defense achieves the highest successful rate compared to Prompt-based defense. This is demonstrated by the experiment (E2) described in Section 8.2 of the paper, with results illustrated in Table 2 of the paper.

A.4.2 Experiments

(E1): Detectors comparison (45 human-minutes + 120 compute-hours (depend on network stability) + 335MB disk):
Preparation: To conduct the comparison experiment, the following steps are executed with precision. Initially, retrieve the essential files as a dataset for PhishLLM, as detailed in Section 6 of the paper, encompassing webpage screenshots, URLs, and HTML source code, with each URL mapped to a distinct folder for storage. Next, establish the experimental environment for PhishLLM by following the guidelines provided in the official repository at "https://github.com/code-philia/PhishLLM". Next, get the test data (Main risk-inclusive response) in Section 6 of the paper.
Execution: Subsequently, execute the PhishLLM method on the collected dataset to obtain results, recording them in a CSV file; Concurrently, apply the HtmlLLM-Detector method using the same dataset and recording them in the same CSV file.
Results: Finally, implement the XGBoost-Detector method through `main.py` and `tools.py`, generating output in JSON format, to derive the confusion matrices for all three detectors and compute the corresponding performance metrics.
(E2): Agent-based and Prompt-based comparison (70 human-minutes + 50 compute-hour (depend on network stability) + 5MB disk):
Preparation: To conduct the comparison experiment, the following steps are executed with precision. Get the test data (Main risk-inclusive response) in Section 6 of the paper.

Execution: Implement the Agent-based approach by executing the script `main.py` with the test data to produce two JSON format files, employing HtmlLLM-Detector and PhishLLM-Detector respectively. Furthermore, separately run `prompt_defense.py` to evaluate the test data and generate an additional JSON format file.
Results: Conduct manual verification by comparing the two JSON format files generated from the Agent-based method with the JSON format files produced by the XGBboost-Detector in experiment E1, alongside the JSON format file generated from independently testing data with `prompt_defense.py`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2025/`.





USENIX Security '25 Artifact Appendix: Generated Data with Fake Privacy: Hidden Dangers of Fine-tuning Large Language Models on Generated Data

Atilla Akkus,[1*] Masoud Poorghaffar Aghdam, [1*] Mingjie Li,[2*] Junjie Chu,[2]
Michael Backes,[2] Yang Zhang,[2] Sinem Sav[1]

[1]*Bilkent University*
[2]*CISPA Helmholtz Center for Information Security*

April 15, 2025

A Artifact Appendix

This artifact appendix provides a comprehensive guide to evaluating the *functionality* of our artifact that includes code, datasets, and evaluation scripts.

A.1 Abstract

This artifact enables the assessment of privacy risks when fine-tuning large language models (LLMs) with generated data. It includes the evaluation of Personal Information Identifier (PII) leakage and Membership Inference Attacks (MIAs), across two fine-tuning strategies: supervised fine-tuning (SFT) with unstructured generated data and self-instruct tuning. The artifact enables researchers to analyse the extent to which generated data may still pose privacy risks. The complete repository and setup instructions are included in this document.

A.2 Description & Requirements

The software environment is defined by the dependencies listed in the requirements.txt file, which includes all necessary Python packages. A pre-configured Docker image, *pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime*, is recommended to ensure compatibility and reproducibility. After setting up the environment, the dependencies can be installed by running: `pip install -r requirements.txt`

A.2.1 Security, privacy, and ethical concerns

We used publicly available data without human participants. Our study examines privacy risks, such as PII extraction and MIA, in fine-tuning LLMs. Findings were responsibly disclosed, and fine-tuned checkpoints were not released due to privacy concerns.

*These authors contributed equally to this work.

A.2.2 How to access

The artifact is available at: https://doi.org/10.5281/zenodo.14732690.

A.2.3 Hardware dependencies

The experiments were conducted on hardware with a CUDA-compatible NVIDIA GPU ($\geq$ 40GB VRAM recommended for training), an x86_64 architecture CPU, at least 32GB of RAM, and 100GB of disk space for storing datasets and intermediate outputs.

A.2.4 Software dependencies

The environment requires Python 3.10+ and packages from 'requirements.txt'. It is suggested to use *'pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime'* docker image.

A.2.5 Benchmarks

We use the Enron subset of the Pile dataset in our paper. The processed version, as used in our study, is included in the repository under the 'data' directory.

A.3 Set-up

A.3.1 Installation

Clone the Repository: Begin by cloning the Git repository to your local machine:

```
git clone [repository URL]
cd [repository directory]
```

Pull the Docker Image:

```
docker pull pytorch/pytorch:2.5.1-
    cuda12.4-cudnn9-runtime
```

Run the Docker Container: Start the Docker container with GPU support and mount the repository:

```
docker run -it --gpus all -v
    \$(pwd):/workspace
pytorch/pytorch:2.5.1-cuda12.4
    -cudnn9-runtime
```

Install Dependencies: Install the required Python packages using the provided `requirements.txt` file:

```
pip install -r requirements.txt
```

A.3.2 Basic Test

Fine-Tuning Models with PEFT

Fine-tune models using the `run_peft.py` script. Example command:

```
torchrun -nproc_per_node=4 run_peft.py -model
EleutherAI/pythia-1.4b
```

Merging Adapters with PEFT

After fine-tuning, use the `merge_peft.py` script to merge the adapter results with the base model:

```
python merge_peft.py -model ./_fine_tuned_model
-save_dir ./merged_model_output
```

To check the functionality of this script, please refer to the checkpoint provided in https://drive.google.com/file/d/1H8_0p5bW8Ur5ZXZHrvOHYsITTzD52Fys/view.

Measuring PII Attack Success

Evaluate PII attack success using the `measure_purified.py` script. Example command:

```
python measure_purified.py -model
EleutherAI/pythia-1.4b -output_dir ./results
```

Evaluating Models with Purified Workflow

Automate merging and PII attack evaluation using the `evaluate_purified.py` script. Example command:

```
python evaluate_purified.py -model ./pythia-1.4b
-temp ./tempMerged
```

Measuring Model Perplexity

Calculate model perplexity using the `perplexity/perplexity.py` script. Example command:

```
python perplexity/perplexity.py -dir ./pythia-1.4b
```

Generating Continuation Data

Generate continuation data using the `data/continuation/generate.py` script. Example command:

```
python data/continuation/generate.py -model
EleutherAI/pythia-1.4b
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Our system includes code for generating synthetic datasets tailored for fine-tuning LLMs.*

(C2): *Our system offers the essential code for fine-tuning models using various Parameter-Efficient Fine-Tuning (PEFT) techniques.*

(C3): *Our system provides the essential code for evaluating models with a streamlined workflow.*

(C3): *Our system provides the essential code for evaluating models with purified workflow.*

(C4): *Our system provides the essential code for measuring the success of PII attacks.*

(C5): *Our system provides the essential code for measuring the success of MIA attacks.*

A.4.2 Experiments

Each experiment listed below directly corresponds to a major claim in Section A.4.1 in sequential order. In addition, several optional parameters are available for each experiment, which are detailed in our GitHub repository.

(E1): *Generating Dataset [1 compute-hour + 1GB disk]*: It generates a synthetic dataset as articulated in Section 3.1.1.
How to: Use the script *data/continuation/generate.py* to generate the needed dataset using a specified model and dataset.
Execution: To run the script, the –data_set parameter must be specified, as it defines the path to the chosen dataset. By default this parameters uses "data/enron.jsonl".
Results: Upon running the script, a new directory will be created in the used working directory with a name similar to the specified model. If the script executes and completes successfully, an Arrow file will be generated within this directory. This file contains the generated data points.

(E2): *Fine-Tuning Models with PEFT [4 compute-hours + 10GB disk]*: This step fine-tunes models using PEFT.
How to: Use the script *run_peft.py* to fine-tune models with PEFT.
Execution: To run the script, the –model parameter must be specified, as it determines the model to be instruction-tuned. By default, this parameter uses "EleutherAI/pythia-1.4b". Additional key parameters include:

- **–input**: Path to the training dataset (Default: "data/enron.jsonl").
- **–peft_type**: PEFT type (Options: lora, dora, pissa; Default: "lora").

Results: Once the script completes, the adapter of fine-tuned model will be stored in the output directory. This adapter needs to be merged with the original model prior to evaluation.

(E3): *Evaluating Models with Purified Workflow [3 compute-hours + 5GB disk]*: This step evaluates models by sequentially running *merge_peft.py* and *measure_purified.py*.
How to: Use the script *evaluate_purified.py* to automate the merging and evaluation of all models in a specified directory.
Execution: To run the script, the –model parameter must be specified, as it defines the path to the model being evaluated. By default, this parameter uses "./pythia-1.4b".

Results: Upon successful execution, a *results_purified.jsonl* file will be generated in each model's directory. This file contains key evaluation metrics, including matches, model name, temperature, and top-k values.

(E4): *Measuring PII Attack Success [1 compute-hour + 2GB disk]*: This step evaluates the success rate of Personally Identifiable Information (PII) attacks.

How to: Use the script *measure_purified.py* to assess the number of successful PII attacks.

Execution: To run the script, the **model** and **output_dir** parameters must be specified, as they define the model being evaluated and the directory where results will be saved.

Results: Upon successful execution, a *results_purified.jsonl* file will be generated in each model's directory. This file contains key evaluation metrics, including matches, model name, temperature, and top-k values.

(E5): *Conducting MIA*: This step conducts MIA in the target models.

How to: Download mimir repository. Then run `run.py --config configs/your_config.json` with your customized configuration file. You can refer to the original repository for reference on configuration details.

Results: Upon successful execution, an output directory will be created, and the summary results will be printed on the console. The directory will contain example plots and data produced during MIA.

A.5 Notes on Reusability

This artifact is adaptable for studying privacy risks in fine-tuning LLMs across different models and datasets. It supports modifications to fine-tuning parameters, attack methods, and evaluation metrics, enabling broader analysis. The modular design allows integration with various models, including instruction tuning and domain-specific adaptations. A pre-configured Docker setup ensures stability and easy extension for future research on privacy-preserving LLM training.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Cloak, Honey, Trap: Proactive Defenses Against LLM Agents

Daniel Ayzenshteyn
Ben-Gurion University, Israel

Roy Weiss
Ben-Gurion University, Israel

Yisroel Mirsky*
Ben-Gurion University, Israel

A Artifact Appendix

A.1 Abstract

The **CHeaT** artifact accompanies our paper ***C**loak, **H**oney, **T**rap: Proactive Defenses Against LLM Agents*. It provides:

- The open-source **CHeaT CLI** for embedding deceptive payloads into text assets.
- **Datasets** compatible with PurpleLlama for reproducing Table 2, Fig. 3, and Fig. 4, as well as the additional dataset `dataset_unicode_honeytokens.json` used specifically for Fig. 2.
- An **evaluation corpus of 11 CTF VMs** used for the end-to-end experiments described in Section 6.2 (Table 3).
- **Token-landmine utilities** (`token-landmines/`) that scan entire vocabularies for rare tokens, trigger hallucinations, and evaluate outputs via GPT-4o.
- An interactive **playground notebook** to load saved *PentestGPT* snapshots, inject new hints/traps, and observe the agent's live reasoning and command stream.

The complete package (~45 GB) is archived on Zenodo (record #15601740) with source mirrored on GitHub.

A.2 Description & Requirements

To reproduce the results from the paper, you will need to use the provided artifact hosted on Zenodo (https://zenodo.org/records/15601740), with source mirrored on GitHub. The artifact includes the complete source code, datasets, and additional evaluation assets necessary to reproduce all experiments described in our paper.

In this artifact, you will find detailed instructions for installation, setup, and usage of the provided tools and datasets. Specifically, the artifact comprises the **CHeaT CLI** tool, evaluation datasets compatible with the PurpleLlama framework, the Unicode honeytoken dataset used for Figure 2, token-landmine utilities, and a Jupyter playground notebook for interactive exploration and experimentation with saved *PentestGPT* snapshots.

*Corresponding Author

The provided evaluation datasets are ready-to-use and correspond exactly to the experiments described in Sections 6.1 and 6.2 of the paper. If you plan to extend this work to evaluate different LLM-based penetration testing agents or experiment with novel payloads, you may need to adjust or regenerate datasets using the methodology described in the paper's experimental sections and the provided **CHeaT CLI** tool.

The provided **token-landmine utilities** were designed specifically for scanning LLM vocabularies, triggering hallucinations, and assessing outputs using GPT-4o. If your experiments involve different LLM architectures or vocabularies, you should rerun these utilities accordingly.

Lastly, while the artifact includes ready-to-run vulnerable CTF VMs, we recommend conducting all experiments strictly within these isolated environments to ensure safety and ethical compliance as detailed in the paper's ethics considerations (Section 9).

A.2.1 Security, privacy, and ethical considerations

All experiments are safe to run on a local machine. However, any evaluation involving autonomous penetration testing agents such as *PentestGPT* must be conducted strictly within the included CTF virtual machines and never against real systems. As detailed in the paper's ethics section, all techniques are purely defensive and designed to mislead malicious agents. The CHeaT tool makes no system-wide changes, operates only on local files, and is fully reversible. The "token-landmine" findings were responsibly disclosed to affected vendors prior to publication and are shared privately with reviewers during the embargo period. Under these constraints, the artifact poses no risk to external systems or user privacy.

A.2.2 How to access

- **Stable archive:** https://zenodo.org/record/15601740
- **Source repository:** https://github.com/Daniel-Ayz/CHeaT
- **Private token-landmine evaluation (review-only):** https://drive.google.com/drive/folders/1YTWdDrkrpaofoH9bAorxN_kR0yQmmPgv

Includes landmine token detection and judgment scripts. This material will be made public after the embargo period (see Section A.2.1).

A.2.3 Hardware dependencies

Most experiments rely on querying LLMs via API and can be run on any standard workstation. Running the CTF virtual machines requires a desktop with basic virtualization support (VT-x/AMD-V) and approximately 50GB of free disk space. For token-landmine detection, local inference depends on the chosen model; our evaluation used an RTX-6000 GPU (24GB VRAM), which sufficed for quantized 70B models. Larger models may require more powerful hardware.

A.2.4 Software dependencies

The CHeaT tool and evaluation scripts require Python 3.9 or later. Dataset-based evaluations rely on the PurpleLlama framework, which should be installed according to its official documentation. Running the CTF virtual machines requires VirtualBox (version 7.0 or higher). To evaluate commercial LLMs such as GPT-4o, an API key (e.g., OpenAI or vendor-specific) is required.

A.2.5 Benchmarks

Datasets are located under `datasets/` for usage with purple llama as documented.

A.3 Set-up

A.3.1 Installation

To install the CHeaT tool, clone the repository (https://github.com/Daniel-Ayz/CHeaT) and follow the instructions in the "Tool Quick Start" section of the main `README.md`. For dataset-based evaluation, install PurpleLlama by following these steps:

1. **Clone the PurpleLlama Repository**

   ```
   git clone https://github.com/
   meta-llama/PurpleLlama.git
   ```

2. **Navigate to the Project and Set Up a Virtual Environment**

   ```
   cd PurpleLlama
   ```

   ```
   python3 -m venv .venv &&
   source .venv/bin/activate
   ```

3. **Install Python Dependencies**

   ```
   pip3 install -r
   CybersecurityBenchmarks/requirements.txt
   ```

4. **Set the Dataset Path**

   ```
   export DATASETS=$PWD/
   CybersecurityBenchmarks/datasets
   ```

You can place our dataset files under a subdirectory such as:

```
$DATASETS/CHeaT/dataset_main.json
```

To run the CTF machines used for end-to-end testing, use the "Import Appliance" option in the VirtualBox GUI to load each `.ova` image and then start the machines.

A.3.2 Basic Test

CHeaT Tool Test: To verify the CHeaT tool is installed correctly, run this basic test from the Tool Quick Start section:

1. Clone the repository and enter the tool folder:

   ```
   git clone https://github.com/Daniel-Ayz/
   CHeaT.git
   cd CHeaT
   ```

2. Create and activate a virtual environment (optional):

   ```
   python3 -m venv .venv && source
   .venv/bin/activate
   ```

3. Install the tool:

   ```
   pip install -e .
   ```

4. Plant a random defense in a test HTML file:

   ```
   echo "<html><body>Hello</body></html>" >
   ./test.html
   ```

   ```
   cheat -action plant -details '{
   "assettype": "web_file",
   "file_path": "./test.html",
   "technique": "random"
   '}'
   ```

5. Verify the installation by listing installed payloads:

   ```
   cheat -action list -type installed
   ```

PurpleLlama Dataset Evaluation Test: For dataset-based evaluations using PurpleLlama, follow the installation instructions provided and refer to the documentation in the `datasets/` directory of the CHeaT repository for specific evaluation procedures with `dataset_main.json`, `dataset_boosted_with_pi.json`, and `dataset_unicode_honeytokens.json`.

CTF Virtual Machine Test: To verify the CTF machines are working correctly:

1. Download the VM files from the Zenodo dataset:
   `https://zenodo.org/records/15601740`

2. Import a CTF machine (e.g., UbuntuX.ova) using VirtualBox's "Import Appliance" option

3. Start the virtual machine and verify it boots successfully

4. Test network connectivity to confirm the machine is ready for evaluation

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our payload techniques achieve a Defense Success Rate (DSR) of 55–67% across all evaluated LLMs. This is supported by Section 6.1 of the paper, using the PurpleLlama-compatible datasets (`dataset_main.json` and `dataset_boosted_with_pi.json`). Results are shown in Table 2 and Figures 3 and 4.

(C2): A defended version of the CTF machines achieves 100% DSR against *PentestGPT* when using CHeaT to insert the top 10 technique–datapoint combinations. This is demonstrated by the experiments in Section 6.2 and shown in Table 3.

(C3): There exist Unicode characters that allow honeytokens to distinguish between human and LLM access, based on mismatches in rendering and interpretation. This is supported by evaluations using `dataset_unicode_honeytokens.json` and illustrated in Figure 2.

(C4): Some rare tokens cause LLMs to produce hallucinations or gibberish when injected. This behavior is identified and measured in our token-landmine study and summarized in Table 11.

A.4.2 Experiments

(E1): *Dataset Evaluation with PurpleLlama* [5 human-minutes + 10 compute-minutes]: Evaluates `dataset_main.json` and `dataset_boosted_with_pi.json` using PurpleLlama to reproduce defense success rates (C1). For detailed instructions, please follow the readme under the `datasets` folder.
Preparation: Install PurpleLlama and copy the datasets from `datasets/` into its `data/` folder.
Execution: Run the prompt-injection benchmark script with the selected dataset and model. Enable `-run-llm-in-parallel` for faster evaluation.
Results: PurpleLlama will generate a summary file reporting Defense Success Rate (DSR) by category, as shown in Figures 3–4 and Table 2.

(E2): *Unicode Honeytoken Evaluation [2 human-minutes + 2 compute-minutes]:* Evaluates `dataset_unicode_honeytokens.json` using PurpleLlama to confirm the behavior difference between human- and LLM-visible tokens (C3).
Preparation: Same setup as in E1. Copy the Unicode dataset into PurpleLlama's `data/` folder.
Execution: Run PurpleLlama's benchmark on the dataset using a target LLM (e.g., GPT-4o).
Results: Inspect the model's responses to see whether the password was misinterpreted. Results are summarized in Figure 2.

(E3): *Token Landmine Evaluation [5 human-minutes + variable compute-time]:* Identifies hallucination-triggering tokens (C4) using the `detect_landmine_tokens.py` and `judge_landmines.py` scripts (in `token-landmines/`). Runtime depends on model size and number of tokens scanned.
Preparation: Follow the usage instructions in the `token-landmines/ README.md`. Ensure the GPU is available for large model scans.
Execution: Run the detection script on the chosen model, then label the responses using the judgment script with an OpenAI API key.
Results: The final JSON includes a judgment label ("Yes"/"No") per token. Results are summarized in Table 11.

(E4): *CTF Machine Evaluation with PentestGPT [1+ human-hours per trial]:* Reproduces full agent-based trials with and without defenses to validate the real-world effectiveness of CHeaT (C2).
Preparation: Import the CTF virtual machines into VirtualBox and launch one. Prepare the environment for *PentestGPT* execution with API access.
Execution:
- Step 1: Run *PentestGPT* on the unmodified CTF machine and solve it using the agent's instructions.
- Step 2: Use CHeaT to insert defenses (e.g., top 10×10 entries) into the same machine.
- Step 3: Repeat the process with *PentestGPT* and observe whether it is blocked or misled.

Results: Compare the actions and success rate of *PentestGPT* between the unmodified and defended versions. Results should match Table 3 (100% DSR).

A.5 Notes on Reusability

The provided CTF machines can serve as a reusable testbed for evaluating future LLM-powered penetration testing agents or novel defense techniques. The datasets and payloads included in the repository offer a way to benchmark whether next-generation agents or models remain vulnerable to the same classes of traps. Additionally, the CHeaT tool is modular and can be easily extended to insert custom payloads, support additional asset types, or implement more sophisticated

deception strategies.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Big Help or Big Brother? Auditing Tracking, Profiling, and Personalization in Generative AI Assistants

Yash Vekaria
UC Davis

Aurelio Loris Canino
UNIRC

Jonathan Levitsky
UC Davis

Alex Ciechonski
UCL

Patricia Callejo
UC3M

Anna Maria Mandalari
UCL

Zubair Shafiq
UC Davis

A Artifact Appendix

A.1 Abstract

This artifact provides a framework for capturing and analyzing the network traffic generated while auditing Generative AI (GenAI) browser assistants. It is designed to assess the tracking, profiling, and personalization in Generative AI (GenAI) browser assistants. These assistants, implemented as browser extensions, offer functionalities such as search-based integration, querying, and page summarization. Leveraging powerful large language models (LLMs), they can access and process a wide range of user data in their browser. We use a MITM-based infrastructure to record network traffic in the form of '.flow' files and then a parser for further analysis of data collection and sharing practices of these assistants.

A.2 Description & Requirements

Our framework comprises capturing and analyzing the network traffic behaviour when using a GenAI extension. The artifact repository consists of the following components:

- **Flows/**: Directory contains the network traffic captured via Mitmproxy in the form of `.flow` files, generated while interacting with GenAI browser extensions. This directory is located in the current working directory and contains sub-directories labeled with extension name being tested (e.g., Merlin).
- **Output/**: Directory contains CSV files generated by processing the captured `.flow` files. Sample CSV files for different extensions generated during the experiments are available in the OSF repository.
- **parse_flows_to_csvs.py**: Python script that parses `.flow` files into summarized CSV files, extracting key details about relevant flows. This includes timestamp, request domain, payload, response type, cookies, domain's parent organization and disconnect list based tracker's category (see Section A.2.5).
- **generate_entity_domain_mapping.py**: Python script that parses individual entity-domain mapping files for each domain from DuckDuckGo's tracker-radar repository and generates a consolidated `ddg.json`.
- **disconnect.json**: Disconnect's tracking protection list used to map domains to predefined tracker categories such as Advertising, Analytics, and Social.
- **ddg.json**: DuckDuckGo's entity-domain list maps the domains to corresponding parent organizations. `ddg.json` is an aggregated single file generated by parsing individual mapping files using `generate_entity_domain_mapping.py`.
- **requirements.txt**: Lists Python package dependencies required to run the analysis pipeline.
- **README.md**: Detailed instructions for setting up Mitmproxy, configuring it in Google Chrome, capturing network traffic, and executing the analysis pipeline.

A.2.1 Security, privacy, and ethical concerns

This artifact involves intercepting HTTPS traffic using a custom root Mitmproxy certificate. Some risk considerations are as follows:

- All HTTPS traffic from the configured browser instance may be logged in a decrypted form by Mitmproxy.
- Evaluators must be careful in choosing what type of online personal or private spaces they login to. This is because in presence of GenAI assistants, user's sensitive information can be potentially extracted and shared with assistant's server or LLM model's server.
- It is recommended to use a fresh Chrome profile and remove the Mitmproxy certificate after experiments.

For more information on ethical considerations, we encourage to read the corresponding section in the main paper.

A.2.2 How to access

The artifact can be accessed via Zenodo at the following link: https://doi.org/10.5281/zenodo.15530229. Additionally, code is also hosted on GitHub repository: https://github.com/Yash-Vekaria/genai-assistants/.

A.2.3 Hardware dependencies

No specialized hardware is required; the artifact can be run on standard consumer machines such as a laptop or a desktop.

A.2.4 Software dependencies

The artifact is compatible with any operating system such as Windows, Linux, and macOS. It requires Python 3.8+, `mitmproxy` for traffic interception, and relevant Python packages as listed in `requirements.txt`. Detailed setup instructions are provided in the README file.

A.2.5 Benchmarks

We use two external resources in our framework:
• *Disconnect List*: allows categorizing tracking domains into one of the 11 categories such as Advertising, Analytics, and Social. It is available at https://github.com/disconnectme/disconnect-tracking-protection/blob/master/services.json.
• *DuckDuckGo's Entity-Domain Mapping*: allows mapping a domain to its parent organization or owner. It is openly available at https://github.com/duckduckgo/tracker-radar/tree/main/domains/US.

A.3 Set-up

The steps listed here allow setting up the necessary environment for evaluating the artifact.

A.3.1 Installation

❶ **MITM Proxy Installation:** Primarily, install `mitmproxy` following the official guide at (https://mitmproxy.org/).
❷ **Instantiate Chrome Profile:** Create a new Google Chrome profile and configure it to route its traffic through the `mitmproxy` server linked to (`localhost:8080`). First, visit chrome://version/ in your Google Chrome browser to figure out path to Chrome executable and/or user data directory. Open Chrome via CLI by specifying a custom user directory.
❸ **Install MITM Certificate:** Now, in the instatiated browser, install the *mitmproxy* root certificate by visiting `mitm.it` in the configured browser and following the prompts to add it to the system's trusted authorities. This step is required only once and ensures that HTTPS traffic can be intercepted.
❹ **Validation:** Once the browser has been configured with `mitmproxy`, validate it by launching the configured Chrome profile, and confirming that intercepted traffic is saved in a `.flow` file.
❺ **Install Python Dependencies:** For the analysis phase, set up a Python virtual environment and install the required Python dependencies.
❻ **Download Benchmarks:** For disconnect list, download `services.json` from here: https://github.com/disconnectme/disconnect-tracking-protection/blob/master/services.json, place the downloaded file inside the current working directory, and rename it to `disconnect.json`. For generating DuckDuckGo's mapping, run `generate_entity_domain_mapping.py` and then place the generated `ddg.json` in the current working directory.

A.3.2 Basic Test

A basic functionality check can be performed to verify the artifact is operational. For the sake of this test, we will consider the browser extension Merlin. Execute the following steps:

First, start `mitmweb` in one terminal and instantiate a fresh or unused chrome profile in the second parallel terminal window. Once the Chrome browser opens, install the GenAI browser extension (in this case Merlin) from the Chrome Web Store. Next, login to the extension and pin it to your browser. Next, perform three interactions: *search*, *browse*, and *summarize*. For *search*, ask a question in Google search and let Merlin auto-generate the response in side space on the right. In case, it does not, but provides an option to generate, click on that option in the side space. Alternatively, extension icon can be clicked to ask a question. For *browse* scenario, visit any webpage and ask a question about the page to the Merlin by clicking on the extension icon. Lastly, for *summarize*, the same webpage can be summarized using one of the extension-provided options. Now close the browser window, and terminate the processes in both the terminal windows.

The successful capture of network traffic can be confirmed by verifying the creation of the `.flow` file in the working directory. The analysis script: `parse_flows_to_csvs.py` can be now run on the captured `.flow` file to generate a summary CSV of the network traffic in the `Output` folder. This can be used to verify different claims in our paper such as: "Merlin can be seen contacting `google-analytics.com` and sharing user's query prompt to this domain in the payload."

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *We visit personal as well as private spaces online in presence of GenAI browser assistant and observe the assistant to collect varying granularity of data such as page's textual content, partial content, form fields, and*

whole DOM depending on the extension as depicted in Table 2. This is demonstrated by experiment (E1) and discussed in Section 5.3 of the paper.

(C2): *An audit of sharing of collected information with first- as well as third-parties is measured for different user-, chat-, and browser-specific identifiers as shown in Table 3 and Figure 3. Sider and Merlin share chat identifiers with* `google-analytics.com` *while TinaMind shares it with* `analytics.google.com`*. Merlin also shares user's raw query prompts with Google Analytics. MaxAI and Harpa also send data to Mixpanel. This can be manually verified with the help of experiment (E2) and is discussed in Section 5.3 of the paper.*

(C3): *We show that some GenAI assistants profile and personalize more than others based on five user attributes: location, age, gender, income, and interests. More specifically, two extensions (Monica and Sider) profile and personalize in-context as well as out-of-context based on all five user attributes. In contrast, Perplexity and TinaMind showed no strong evidence of profiling or personalization, while Harpa exhibited only in-context personalization behavior. The results in Table 4 can be reproduced by performing experiment (E2) as discussed in Section 5.4 of the paper.*

A.4.2 Experiments

Our auditing framework is semi-automated, with the data collection infrastructure being automated but the experiments being manual.

(E1): *Auditing User Tracking (Data Collection) in GenAI Assistants [30 human-mins. for experimentation + 15 human-mins. for manual analysis per extension]:*

Preparation: To reproduce results in Table 2, read Section 5.3 and refer to prompts in Table 6. Since the experiments were performed and analyzed manually, the evaluators can limit the evaluation to at most 20 scenarios, comprising of public as well as private spaces described in Table 2. Create fresh accounts for each of the 10 private spaces (wherever possible). Ensure that the crawling infrastructure is set-up and validated as explained in Section A.3 and account for the assistant being tested is pre-created. In ChatGPT for Google, it also needs to be linked to a `chatgpt.com` account.

Execution: To test the selected extension, for each space, follow Section A.3.2 to start `mitmweb` in one terminal window and initiate a new browser profile in the second terminal. Within the browser instance, install the extension, pin it, and login to extension. Now, visit the space being tested and open some sensitive or copyrighted content to understand collection practices of the assistant. If it's a private space, first authenticate into the website and then navigate to an appropriate content page to test for. Refer to Table 6 row corresponding to the selected space and ask those prompts in the sidebar chat of the assistant one after the other. Once the testing completes, close the browser instance and stop the data collection to save the `.flow` file.

Results: Manually analyze the flow file of each or a sample of evaluated spaces to validate the observed data sharing with respect to the results displayed in the Table 2 for the tested extension.

(E2): *Auditing User Tracking (Data Sharing), Profiling, and Personalization in GenAI Assistants [30 human-mins. for experimentation + 15 human-mins. for manual analysis per extension]:*

Preparation: Similar to E1, we manually perform three scenarios for each extension – search, browse, and summarize. Read Section 5.4 to understand how to perform experiments for each scenario using fresh accounts (wherever possible). If reusing the same account (e.g., Google account), ensure all data is deleted between subsequent tests and that each test is carried out in a fresh browser profile.

Execution: For each scenario per extension, use a distinct browser profile and start `mitmweb` and instance of the browser from two separate terminal windows, followed by installing the extension and performing the login. Next, perform each experimental scenario as per Section 5.4 of our paper using the prompts listed in Section 8.1. Table 5 (Section 8.3) lists the webpages to visit during the browse and summarize scenarios. After experimenting all scenarios of an extension, generate a summarized .csv file.

Results: The generated `.csv` file can be used to compare the data sharing practices of the assistant with the first- and third-parties by analyzing payload, request URL, and cookies against Table 3. Likewise, third-party sharing flows in Figure 3 can also be easily verified. Profiling and personalization of the selected assistant across the three scenarios can be validated against Table 4 by analyzing the responses returned by the assistant while performing experiments.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2025/`.



USENIX Security '25 Artifact Appendix: Private Investigator: Extracting Personally Identifiable Information from Large Language Models Using Optimized Prompts

Seongho Keum[†] Dongwon Shin[†] Leo Marchyok[‡] Sanghyun Hong[‡] Sooel Son[†]
[†]KAIST [‡]Oregon State University

A Artifact Appendix

A.1 Abstract

Private Investigator is an attack framework designed to optimize prompts for querying a Language Model (LM) in order to extract Personally Identifiable Information (PII) used during its fine-tuning process. This artifact provides Docker image to setting up the experimental environment, along with Python programs required to reproduce the results presented in the paper. We tested the artifact on an Ubuntu 22.04 machine equipped with an NVIDIA GeForce RTX 3090 GPU. The experiments require 24 GB of GPU memory, although they can be executed on GPUs with smaller memory capacity by appropriately reducing the batch size. We expect this artifact to reproduce the evaluations shown in §5.2, Table 2, and Figures 2, 10, 11, and 12 of the paper. However, please note that the exact experimental results may vary due to the probabilistic nature of text generation in large language models.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Private Investigator does not have any kind of security risks to the machines that execute it. Furthermore, all the datasets used for the experiments are publicly accessible.

A.2.2 How to access

The artifact is accessible through GitHub (https://github.com/WSP-LAB/PrivateInvestigator) and Zenodo (https://doi.org/10.5281/zenodo.16748364).

A.2.3 Hardware dependencies

Private Investigator requires a Linux machine with an NVIDIA graphics card. Due to the large size of LMs and datasets, we recommend using a machine with at least 32 CPU cores, 128 GB of system memory, 24 GB of GPU memory, and 150 GB of disk space. We tested the Docker image and Python programs on a machine running Ubuntu 22.04 with an NVIDIA GeForce RTX 3090 GPU.

A.2.4 Software dependencies

Private Investigator requires CUDA 12.1 and Python 3.10, along with the following Python libraries: Pytorch 2.1.2, Transformers 4.51.3, and Flair (https://github.com/flairNLP/flair).

The artifact utilizes two datasets (Enron and TREC) and four LM architectures (GPT-2, GPT-Neo, OpenELM, and PHI-2). The Enron dataset can be accessed at https://www.cs.cmu.edu/~enron/, and the TREC dataset is available at http://curtis.ml.cmu.edu/w/courses/index.php/W3C_Email_Corpus. All four LM architectures are publicly shared on Hugging Face (https://huggingface.co/models). All architectures are accessible without any authorization. However the Llama-2 tokenizer–used by the OpenELM model–requires a model access token, which can be obtained from its Hugging Face page: https://huggingface.co/meta-llama/Llama-2-7b. We also share the weights of LMs fine-tuned on the Enron and TREC datasets on Hugging Face: https://huggingface.co/SeonghoKeum.

Instructions for downloading the required software, datasets, and model weights are provided in § A.3.1.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

1. Download the artifact from our GitHub repository: https://github.com/WSP-LAB/PrivateInvestigator.

2. Install Docker and configure it to run Docker without root privileges.

3. Install CUDA Toolkit 12.1. If different version of CUDA is used, update the DockerFile accordingly.
4. To enable GPU access within Docker container, install the NVIDIA Container Toolkit.
5. Build the Docker container by running the script: `build.sh[MODEL_ACCESS_TOKEN`.
6. Launch the Docker container by running the script `launch_container.sh`.
7. Download the datasets by executing the following scripts: `get_enron.sh`, `get_trec.sh`, and `get_commoncrawl.sh`.
8. Download fine-tuned model weights by running the scripts: `get_model_weights.sh` and `get_model_weights_counter.sh`.

For more details, please refer to the `README.md` file included in the artifact.

A.3.2 Basic Test

To verify that all required software dependencies are properly installed, run `basic_test.py` located in the `experiments/` directory. This script specifically tests the TREC dataset and evaluates its perplexity on the OpenELM model fine-tuned on that dataset. If everything is set up correctly, it should print a perplexity value of 3.6.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Private Investigator extracts more PIIs compared to the baselines in 17 out of 24 cases. This is demonstrated by the experiments (E2) and (E3). These experiments are descibed in §5.1 and the results are reported in Table 2 of our paper.

(C2): Private Investigator extracts exclusive PIIs that are not extracted by any other baselines. The claim is proven by Experiment (E4). This experiment is described in §5.2 and illustrated in Figures 2, 10, 11, and 12 of the paper.

A.4.2 Experiments

All scripts and Python files for conducting the experimeents are located in the `experiments/` directory of the artifact. In this section, the directory is omitted in the paths.

(E1): Prompt Generation [600 GPU-hours]: Private Investigator generates prompts and store them under `prompts/` directory.
Preparation: Download all required datasets and model weights, and complete the environemnt setup as described in § A.3.1.
Execution: Run the Python program `run_generate_prompt.py` with the appropriate arguments. It requires the following arguments:
- architecture: gptneo
- model_ckpt: weights/gptneo-enron, weights/gptneo-trec
- surrogate: fine, pre
- target: email, phone, name
- from_scratch: True, False

To generate prompts using the pre-trained surrogate model, omit the model_ckpt argument.
Generating prompts from scratch for all combinations of datasets, surrogate models, and target PII types takes approximately 600 GPU-hours. To reduce computational cost, the program is designed to use intermediate text generation results stored in the `text_generation/` directory by default, which allows prompt generation to complete within an hour. To generate prompts from scratch, explicitly set the from_scratch argument to True. To generate prompts for all combinations at once, use the following scripts: `run_generate_prompt_interim.sh` (uses intermediate results) `run_generate_prompt_scratch.sh` (generates from scratch).
Results: Generated prompts will be saved in the `prompts/` directory.

(E2): Extract PIIs [200 GPU-hours]: Private Investigator extracts PIIs using the generated prompts.
Preparation: Complete experiment (E1).
Execution: Execute the Python program `run_attack_campaign.py` with the required arguments:
- architecture: gpt2, gptneo, openelm, phi2
- model_ckpt: weights/[gpt2 | gptneo | openelm | phi2]-[enron | trec]
- surrogate: pre, fine
- target: email, phone, name
- c: 0.25
- temp: 1 (for email and phone), 1.4 (for name)

Adjust the eval_batch_size argument to balance GPU memory usage and execution time, if needed.
To run Private Investigator across all combinations of architectures, datasets, and PII types, execute the `run_extract_piis_all.sh` script.
Results: The number of extracted PIIs will be saved in the `all_attack_result.txt` file. These results will be similar to those reported in the Table 2 of the paper. Note that the exact numbers may vary due to the non-deterministic behavior of the LMs.

(E3): Baselines [400 GPU-hours]: Run four baseline attacks: Carlini *et al.* (Top-K), Carlini *et al.* (Temp), Carlini *et al.* (Internet), and Lukas *et al.*.
Preparation: Download all required datasets and model weights, and complete the environemnt setup as described in § A.3.1.

Execution: To run the Carlini *et al.* baselines, execute `baseline_carlini.py` with the following arguments:

- model_path: weights/[gpt2 | gptneo | openelm | phi2]-[enron | trec]
- method: topk, temperature, internet
- target: email, phone, name
- temp: 1 (for email and phone), 1.4 (for name)

The batch_size argument can be adjusted to balance GPU memory usage and execution time.
To runt the Lukas *et al.* attack, execute `baseline_lukas.py` with the following four arguments:

- architecture: gpt2, gptneo, openelm, phi2
- model_ckpt: weights/[gpt2 | gptneo | openelm | phi2]-[enron | trec]
- target: email, phone, name
- temp: 1 (for email and phone), 1.4 (for name)

Similar to the previous attacks, generation batch size could be controlled by the eval_batch_size argument.
To run all experiments at once, execute the following scripts: `run_baselines_email.sh`, `run_baselines_phone.sh`, and `run_baselines_name.sh`.
Results: All baseline results will be saved in the `all_attack_result.txt` file. Note that the results may not exactly match those reported in the paper due to the probability based text generation.

(E4): PII overlap [less than one hour]: Count the number of PIIs that are exclusively or commonly extracted by each attack.
Preparation: Complete the experiments (E1), (E2), and (E3).
Execution: Execute `analyze_pii_overlap.py`.
Results: The numbers of exclusively and commonly extracted PIIs will be saved in the `overlap.txt` file.

Instructions for additional analysis experiments are provided in the `README.md` file included in the artifact.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: JBShield: Defending Large Language Models from Jailbreak Attacks through Activated Concept Analysis and Manipulation

Shenyi Zhang[1], Yuchen Zhai[1], Keyan Guo[2], Hongxin Hu[2], Shengnan Guo[1], Zheng Fang[1], Lingchen Zhao[1], Chao Shen[3], Cong Wang[4], and Qian Wang[1]
[1] Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University,
[2] University at Buffalo, [3] Xi'an Jiaotong University, [4] City University of Hong Kong

A Artifact Appendix

A.1 Abstract

This artifact supports the evaluation of JBShield, a comprehensive jailbreak defense framework for large language models (LLMs). JBShield leverages the Linear Representation Hypothesis (LRH) to investigate the mechanisms underlying jailbreak attacks, defining and analyzing high-level toxic and jailbreak concepts encoded in LLM hidden representations. By identifying and manipulating these concepts, JBShield effectively detects and mitigates jailbreak prompts, ensuring safer outputs. This artifact includes: (1) datasets used for evaluation, encompassing nine types of jailbreak prompts across five target LLMs, benign prompts from the Alpaca dataset, and harmful prompts from AdvBench and Hex-PHI; (2) implementations for extracting concept-related interpretable tokens; and (3) testing scripts for evaluating JBShield-D (detection) and JBShield-M (mitigation) across five LLMs. These resources facilitate reproducibility and provide a comprehensive assessment of JBShield's effectiveness in reducing attack success rates and enhancing LLM safety.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our research focuses on defending jailbreak attacks on large language models (LLMs). As such, the provided dataset contains examples of harmful language, and the evaluation process may involve executing jailbreak attacks that could lead the model to generate ethically or morally questionable content. While these outputs are used strictly for research and defense evaluation purposes, reader discretion is recommended. Evaluators should take necessary precautions when reviewing the dataset and generated outputs to ensure responsible handling of potentially harmful content.

A.2.2 How to access

The artifact is publicly available and can be accessed at the newest version under the Zenodo URL https://zenodo.org/records/14732884 and GitHub repo https://github.com/NISPLab/JBShield. This repository contains all necessary components for evaluating JBShield, including the datasets, code for extracting concept-related tokens, and implementations of JBShield-D (detection) and JBShield-M (mitigation) for testing across five target LLMs.

A.2.3 Hardware dependencies

Since our research focuses on evaluating large language models (LLMs), the primary hardware requirement is a GPU with sufficient memory to handle inference and jailbreak detection tasks. The minimum hardware requirement is two GPUs with at least 24GB VRAM each (e.g., RTX 3090 or RTX 4090). For optimal performance, we recommend a setup with 4 RTX 4090 GPUs (24GB VRAM each) or 1 A100 GPUs (80GB VRAM each). These configurations ensure efficient execution of JBShield's detection and mitigation components across multiple LLMs.

A.2.4 Software dependencies

We recommend evaluating our artifact on a Linux-based system. Our implementation has been tested and verified on Ubuntu 20.04.2 with Linux kernel versions 5.15.0-130-generic and 5.15.0-125-generic. The artifact requires Python for execution, along with several third-party libraries. We suggest following the official PyTorch installation guide[1] to set up PyTorch, while other dependencies can be installed using Pypi. Additionally, our evaluation utilizes five target LLMs and one evaluation LLM, all of which can be downloaded from HuggingFace. For further details, please refer to the README file included in the artifact.

[1] https://pytorch.org/get-started/locally

A.2.5 Benchmarks

The dataset required for the experiments reported in our paper is fully included in the artifact (in *./data*). It comprises harmful prompts (*./data/harmful{}.csv*), benign prompts (*./data/harmless{}.csv*) and jailbreak prompts (in *./data/jailbreak*). Harmful prompts sourced from AdvBench and HEx-PHI datasets. Benign prompts sampled from the Alpaca dataset. Jailbreak prompts generated using nine different jailbreak attack methods, targeting five different LLMs. All jailbreak prompts were generated by our own. The dataset is structured and readily accessible within the artifact for ease of use in reproducing our experimental results. Note that our mitigation evaluation is conducted using 50 samples per jailbreak attack. Specifically, for transfer-based attacks (IJP, Puzzler, Zulu, and Base64) that do not directly exploit target LLM information, we randomly select 50 corresponding jailbreak prompts from our dataset to assess the attack success rate (ASR). For the other jailbreak methods, we treat the defended model as a new target LLM, generate 50 new jailbreak prompts, and compute the ASR accordingly. The evaluation data for jailbreak mitigation is provided in *./data/mitigation*, following the same file structure as *./data/jailbreak*.

A.3 Set-up

A.3.1 Installation

The code for JBShield runs with Python 3 and requires Pytorch. We recommend using Anaconda or miniconda for python. Our code has been tested with python=3.12.8 and torch=2.5.1 on linux. First, install conda[2] (if not already installed), create a conda environment and activate it. Follow the official PyTorch installation guide to install Pytorch and pip install the other packages. Than download and set up all models required. For further details on installation and configuration, please refer to the README file included in the artifact.

A.3.2 Basic Test

To verify that the installation is successful and all required software components are functioning correctly, Run a sample Concept Extraction test in shell by

```
python interpret.py –model mistral
```

Expected output is as

```
[nltk_data] Downloading package words to <path>...
[nltk_data] Package words is already up-to-date!
Number of harmful prompts: 850
Number of harmless prompts: 850
Number of gcg prompts: 850
Number of autodan prompts: 850
Number of saa prompts: 850
Number of drattack prompts: 520
Number of pair prompts: 850
Number of puzzler prompts: 50
Number of ijp prompts: 850
Number of base64 prompts: 850
Number of zulu prompts: 850
Number of harmful prompts: 850
Number of harmless prompts: 850
mistral Interpretation done.
```

[2] https://docs.anaconda.com/anaconda/install

A.4 Evaluation workflow

A.4.1 Major Claims

The major claims made in your paper are enumerated as follows:

(C1): Our method for extracting concept-related interpretable tokens enables a deeper understanding of jailbreak mechanisms by identifying toxic and jailbreak concepts encoded in LLM representations. This is demonstrated through in Section 3 (Activated Concept Analysis), with supporting results shown in Table 2 and Appendix A.

(C2): JBShield-D accurately detects jailbreak prompts by analyzing the activation of both toxic and jailbreak concepts. The effectiveness of JBShield-D is validated by the experiments in Section 5.4 (Jailbreak Detection), where it achieves an average detection accuracy of 0.95 across five LLMs. The results are presented in Table 4.

(C3): JBShield-M successfully mitigates jailbreak attacks by adjusting the hidden representations of LLMs, enhancing toxic concepts while suppressing jailbreak concepts. The effectiveness of JBShield-M is evaluated in Section 5.5 (Jailbreak Mitigation), demonstrating a reduction in jailbreak attack success rates from 61% to 2% across five LLMs. The results are detailed in Table 7.

A.4.2 Experiments

To evaluate JBShield and validate our key claims, we conduct the following experiments. Each experiment is associated with one of the major claims listed in Section A.4.1 and provides details on the execution steps, expected runtime, and required resources.

(E1): [Concept-related Interpretable Token Extraction] [~1 compute-hour]: validate that JBShield's extraction of concept-related interpretable tokens can help in understanding jailbreak mechanisms and identifying toxic and jailbreak concepts.

How to: Extract concept-related tokens from five target LLMs using JBShield's interpretation module.
Preparation: Ensure that the environment and models is set up following the installation instructions in Section A.3. Verify that calibration and test datasets are properly structured in $./data/jailbreak$ as mentioned in README.
Execution: Run the interpretation script:

```
chmod +x ./interpret.sh
./interpret.sh
```

Results: The extracted tokens for each model are saved in $./interpret_results/\{model_name\}.txt$. These results should align with the findings presented in Table 2 and Appendix A of the paper.

(E2): [Jailbreak Detection Performance of JBShield-D] [~4 compute-hour]: evaluate JBShield-D's jailbreak detection performance across five LLMs and verifies that the detection achieves high accuracy and F1-Score.
How to: Run the jailbreak detection module on the five target models and evaluate detection accuracy and F1-Score.
Preparation: Ensure that the environment and models is set up following the installation instructions in Section A.3. Verify that calibration and test datasets are properly structured in $./data/jailbreak$ as mentioned in README.
Execution: Run the detection script:

```
chmod +x ./evaluate_detection.sh
./evaluate_detection.sh
```

Results: The detection performance logs are saved in $./logs/JBShield-D_\{model_name\}.log$. Expected results should match Table 4 in the paper.

(E3): [Jailbreak Mitigation Performance of JBShield-M] [~2 compute-hour]: evaluates JBShield-M's ability to mitigate jailbreak attacks and reduce the attack success rate.
How to: Apply JBShield-M to the five target models and measure the attack success rate.
Preparation: Ensure the detection module is functioning correctly by verifying results from (E2).
Execution: Run the mitigation script:

```
chmod +x ./evaluate_mitigation.sh
./evaluate_mitigation.sh
```

Results: The mitigation results are saved in /logs/JBShield-M.log. Expected results should align with Table 7 in the paper, showing an attack success rate reduction from 61% to 2%.

A.5 Notes on Reusability

Please note that the experimental results presented in this artifact may exhibit some variations due to differences in testing environments, the randomness in calibration set selection, and dataset size discrepancies across different jailbreak methods (DrAttack and Puzzler contain fewer samples compared to others). Despite these variations, the overall trends and effectiveness of JBShield remain stable, as demonstrated by the extensive evaluations conducted in our study.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Web Execution Bundles: Reproducible, Accurate, and Archivable Web Measurements

Florian Hantke
CISPA Helmholtz Center for Information Security

Peter Snyder
Brave Software

Hamed Haddadi
Imperial College London & Brave Software

Ben Stock
CISPA Helmholtz Center for Information Security

A Artifact Appendix

Recently, reproducibility has become a cornerstone in the security and privacy research community, including artifact evaluations and even a new symposium topic. However, Web measurements lack tools that can be reused across many measurement tasks without modification, be robust to circumvention, and remain accurate across the wide range of behaviors in the Web. As a result, most measurement studies use custom tools and varied archival formats, each of unknown correctness and significant limitations, systematically affecting the research's accuracy and reproducibility.

To address these limitations, we present WebREC, a Web measurement tool that is, compared against the current state-of-the-art, *accurate* (i.e., correctly measures and attributes events not possible with existing tools), *general* (i.e., reusable without modification for a broad range of measurement tasks), and *comprehensive* (i.e., handling events from all relevant browser APIs). We also present *.web*, an archival format for the accurate and reproducible measurement of a wide range of past website behaviors. We empirically evaluate WebREC's accuracy, by replicating well-known Web measurement studies and showing that WebREC's results more accurately match our baseline. We then assess if WebREC and *.web* succeed as general-purpose tools, which could be used to accomplish many Web measurement tasks without modification. We find that this is so: 70% of papers discussed in a 2024 web crawling SoK paper could be conducted using WebREC as is, and a larger number (48%) could be leverage against *.web* archives without requiring any new crawling.

A.1 Abstract

The goal of the artifacts evaluation is to see if you are able to work with our tool of if there are any issues we can improve. Therefore, we would like you to focus on testing and playing around with our tool.

To make it clear, all the functionality of the tool we are talking about in the paper (WebREC), is implemented in *pagegraph-crawl* [1], a tool we continuously maintain. We decided to do so to keep the anonymity during the peer-reviewing process. A tool to analyze the archives is *pagegraph-query* [2]. As mentioned, we would like you to focus on these two and tell us if there is anything to improve that would help you to get easier started with it.

Additionally, we provide the code used in this paper in on Zenodo [3]. This includes the pipeline to run pagegraph-crawl together with the two proxies (mitmdump and warcprox) mentioned in the paper, scripts to replay responses from the archives, and scripts for analysis.

A.2 Description & Requirements

This section list all the information necessary to recreate the same experimental setup we used to test the artifact.

A.2.1 Security, privacy, and ethical concerns

Please make sure not to spam any website with the crawler. Other than that, we see no security, privacy, and ethical concerns.

A.2.2 How to access

You can access the code under the following repositories:

- pagegraph-crawl: https://github.com/brave/pagegraph-crawl
- pagegraph-query: https://github.com/brave-experiments/pagegraph-query
- Analysis pipeline: https://doi.org/10.5281/zenodo.15091772

[1] https://github.com/brave/pagegraph-crawl
[2] https://github.com/brave-experiments/pagegraph-query
[3] https://doi.org/10.5281/zenodo.15091772

A.2.3 Hardware dependencies

We developed our pipeline under MacOS and run it on a Ubuntu 22.04.4 LTS server. We do not expect any specific hardware needed for our tools.

A.2.4 Software dependencies

The code requires Python (tested with 3.10.12) and Node.js (tested with 20.11.1) to be installed. If you run the code on a server, a virtual X-server like `xvfb` might be needed. The repositories include a `requirements.txt` for Python dependencies, which can be installed using `pip install -r requirements.txt`, and a `package.json` for Node.js dependencies, installed via `npm install`.

Additionally, Brave Nightly, the testing and development version of Brave needs to be install. Follow the instructions at: https://brave.com/download-nightly/

A.2.5 Benchmarks

None

A.3 Set-up

This section contains all the installation and configuration steps required to prepare the environment to be used for artifact evaluation.

A.3.1 Installation

Below are the installation steps for all repositories. Each step is independent and should start from your initial working directory.

Pagegraph Crawl. To install the crawler, clone the repository and build it with `npm`.

```
git clone
 ↪ git@github.com:brave/pagegraph-crawl.git
cd pagegraph-crawl
npm install
npm run build
```

Pagegraph Query. To use the query tool, clone the repository and install the Python requirements.

```
git clone git@github.com:brave-experiments/
 ↪ pagegraph-query.git
cd pagegraph-query
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

If you use a Python version before 3.11, you will get an error that StrEnum is missing. You can fix this by installing *StrEnum* (`pip install StrEnum`) and adding `from strenum import StrEnum` at the top of the types file (`pagegraph-query/pagegraph/types.py`). Also, you need to remove `StrEnum` as import from `enum` in line 4.

Experiment Pipeline. To setup the experiment pipeline described in Section 4 to collect page graph, HAR, and WARC files, and replays responses from HAR and WARC, follow the steps below. Clone the repository, set up a virtual environment, and install dependencies. Due to version conflicts, in addition to the explained `.venv` you will need a second `.venv_warc` installing `requirements_warc.txt` to run `replay_warc.py` later. All requirements for the analysis scripts can be installed with `requirements_analysis.txt`, we recommend a `.venv_analysis` environment.

```
git clone https://github.com/cispa/WebREC.git
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
cd src
```

Now, you still need to create a config file `config.py` with the following template:

```
TELEGRAM_API_KEY = 'secret'
TELEGRAM_CHAT_ID = 1
JS_HOOKING = True
SCRIPT_PATH = "./js_injections/js_hooks.js"
INITIALIZATION_BREAK = 60
BRAVE_EXEC_PATH = "/opt/brave.com/brave-
 ↪ nightly/brave-browser-nightly"
```

After creating the config, you are able to run the init command. It loads and installs the additionally needed tools such as pagegraph-crawl.

```
python main.py --init --output ./output
```

If you want to test against sites that deploy a CSP, you need to patch pagegraph crawl with `await page.setBypassCSP(true)` as described in the readme of our repository.

If you want to test running in multiple threads, we also recommend to path pagegraph crawl with xvfb retires as described in the readme of our repository.

Analysis via Jupyter Notebooks For the analysis files, we use Jupyter Notebooks. There are various ways to execute these files. One way would be to open it in VSCode which automatically shows options to execute it. As an alternative, you can use the Web base version as follows:

1. Install Jupyter Notebook (if not already installed):

   ```
   pip install notebook
   ```

2. Navigate to the directory containing your notebook:

   ```
   cd /path/to/your/notebook
   ```

3. Launch Jupyter Notebook:

   ```
   jupyter notebook
   ```

4. Open the `.ipynb` file in your browser and run the cells.

A.3.2 Basic Test

This section provides instructions to perform simple functionality tests for the tools. Each test should run in the individual repositories you previously cloned.

Pagegraph Crawl. To test if the crawler is built, you can run `npm run crawl -- -h` to get a list of commands. As a next test, you can run the first crawl as described below.

```
# Working directory ./pagegraph-crawl
npm run crawl -- -u "https://example.org" -t 5 -o output/ --har --har-body --screenshot

ls output
page_graph_https___example_org__1736505833.graphml
page_graph_https___example_org__1736505833.png
page_graph_https___example_org__1736505833.har
```

Upon successful execution, the `output` directory will contain:

- A pagegraph file (`.graphml`) representing the page behavior.
- A screenshot of the website (`.png`).
- A HAR file (`.har`) capturing all requests made within the page's context.

If the tool cannot detect your installed Brave Nightly browser, specify its path manually using: `-b /Applications/Brave Browser Nightly.app/Contents/MacOS/Brave Browser Nightly`

Pagegraph Query. To verify the installation, run `python run.py -h` to list available commands. Then validate the output of a previous crawl with the following command:

```
# Working directory ./pagegraph-query
python run.py validate ../pagegraph-crawl/output/page_[...].graphml

{"meta": {"versions": {"tool": "0.9.5", "graph": "0.7.1"}, "url": "https://example.org/"}, "report": {"success": true}}
```

Experiment Pipeline. To test and familiarize yourself with the experiment pipeline, run `python main.py --help` to view available options. To perform a test crawl, execute:

```
# Working directory ./webrec
cd src
python main.py --output ./output --workers 1 --origins https://example.org
```

Upon successful execution, the `output/` directory will contain a timestamped folder with a subdirectory named `https_example.org` holding all the generated files and archives. If the execution fails, retries will be stored in directories such as `https_example.org_failed_attempt_X`. Check `output/<time>/<domain>/logs/` for detailed logs to debug issues.

If you try to run the pipeline inside a docker, it might be that you need to turn of the sandbox mode by passing the `--no-sandbox` option via the `-extra-args` in `src/misy.py:182`.

A.4 Evaluation workflow

This section includes all the operational steps and experiments which can be performed to evaluate the functionality of our artifacts.

A.4.1 Major Claims

The major claims we make are the following:

(C1): Pagegraph-crawl can generate a graph representing page behavior, a HAR file and a screenshot for any webpage in its page context. Pagegraph-query can analyze the behavior graph. To try this out, follow experiment E1.

(C2): The pipeline described in Sec. 3 for the project generates page behavior graphs, HAR, and WARC files. It also allows to replay from HAR and WARC generating the datasets for the replay experiments. Experiment E2 verifies this.

(C3): The pipeline can be used to identify and measure pages that use JavaScript-added inline event handlers that would violate *unsafe-inline*. Experiment E3 demonstrates this.

(C4): The pipeline can be used to compare the number of HTTP requests between different archiving file formats. Experiment E4 demonstrates this.

(C5): The pipeline can be used to compare the appearances of JavaScript executions between different archiving file formats. Experiment E5 demonstrates this.

A.4.2 Experiments

This section describes the experiments to prove the claims from the previous section.

(E1): [Pagegraph Crawl and Query] [X human-minutes]: As first experiment, we would like you to play around with `pagegraphl-crawl` and `pagegraphl-query` and give us feedback if you had any problems setting it up or running it. Take as much time as you want.

(E2): [Experiment Pipeline] [30 human-minutes]: In this experiment, please test if you get the experiment pipeline running to collect pagegraph, HAR, and WARC files and replay from HAR and WARC.
How to crawl: With the `.venv` environment, run `python main.py --output ./output --workers 1 --origins https://example.org https://google.com` or use your list of origins.
How to replay HAR: With the `.venv` environment, run `python replay_har.py --initial-crawl output/timestamp/` with *timestamp* being the one of your previous crawl.
How to replay WARC: With the `.venv_warc` environment, run `python replay_warc.py --initial-crawl output/timestamp/` with *timestamp* being the one of your previous crawl.
Results: Verify the output directory contains one timestamp directory with subdirectories for each visited domain. Each domain directory should include `.graphml`, `.har`, and `.warc` files, along with replay directories for HAR (`mitmd_replay`) and WARC (`warc_replay`).

(E3): [Event Handler Experiment] [15 human-minutes]: Run a simple version of the Event Handler (Document Object Model) experiment from the paper to validate the pipeline's detection of JavaScript-added inline event handlers.
Preparation: Change the `config.py` as follows:

```
JS_HOOKING = True
SCRIPT_PATH =
↪ "./js_injections/js_hook_smurf_new.js"
```

Then, run the test page in `examples/` on your local machine, for example with `python3 -m http.server`.
How to: Run the experiment pipeline for this page: `python main.py --output ./output --workers 1 --origins http://localtest.me:8000` and generate the WARC and HAR replay directories as in E2. Important: Do not use localhost, but a domain like *localtest.me* that points to localhost. Otherwise, the traffic bypasses the proxies.
For the analysis, change in the `analysis/` directory and the `.venv_analysis` environment. Then execute `event_handler.py`. Change `CRAWL_PATH` in line 18 of this file to point to your crawl. To see the results, open and run `csp_analysis.ipynb` with the produced CSV.
Results: The result should show that PG, Smurf, HAR Smurf, WARC Smurf and All Smurf_1 all have found one origin with an JavaScript-added inline event handlers that would violate *unsafe-inline*.

(E4): [Resources Experiment] [15 human-minutes]: Run a simple version of the Requested Resources experiment from paper to compare the number of resource requests across archiving formats.
Preparation: Change the `config.py` as follows:

```
JS_HOOKING = True
SCRIPT_PATH = "./js_injections/js_hooks.js"
```

Then, run the test page in `examples/` on your local machine, for example with `python3 -m http.server`.
How to: Run the experiment pipeline for this page: `python main.py --output ./output --workers 1 --origins http://localtest.me:8000`. This time, you do not need replays. Important: Do not use localhost, but a domain like *localtest.me* that points to localhost. Otherwise, the traffic bypasses the proxies.

For the analysis, change in the `analysis/` directory and the `.venv_analysis` environment. Then execute `requests_analysis.py`. Change `CRAWL_PATH` in line 16 of this file to point to your crawl. To see the results, open and run `requests_analysis.ipynb` with the produced CSV `requests_results_timestamp_blocklist.csv`.
Results: The result should show that PG and HAR have the same number of requests. Additionally, you can see that the page loaded the 3rd party tracker *googletagmanager.com* blocked by the easyprivacy list.

(E5): [JS Executions Experiment] [15 human-minutes]: Run a simple version of the JavaScript experiment from paper and compare JavaScript execution counts across different archiving formats. In order for you to not compile your own browser, we added a hook for `window.setTimeout` that the test page uses.
Preparation: Change the `config.py` as follows:

```
JS_HOOKING = True
SCRIPT_PATH = "./js_injections/
 ↪ js_hooks_js_experiment.js"
```

Then, run the test page in `examples/` on your local machine, for example with `python3 -m http.server`.
How to: Run the experiment pipeline for this page: `python main.py --output ./output --workers 1 --origins http://localtest.me:8000` and generate the WARC and HAR replay directories as in E2. Important: Do not use localhost, but a domain like *localtest.me* that points to localhost. Otherwise, the traffic bypasses the proxies.
For the analysis, change in the `analysis/` directory and the `.venv_analysis` environment. Then execute `js_compare.py`. Change `CRAWL_PATH` in line 17 of this file to point to your crawl. To see the results, open and run `js_analysis.ipynb` with the produced CSV `js_results_timestamp.csv`.
Results: The result should show that PG, HAR, and WARC have the same number (1) of appearances: *Equal number of appearances: 1 / 1 (1.0).*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.



USENIX Security '25 Artifact Appendix: The DOMino Effect: Detecting and Exploiting DOM Clobbering Gadgets via Concolic Execution with Symbolic DOM

Zhengyu Liu, Theo Lee, Jianjia Yu, Zifeng Kang, and Yinzhi Cao
{zliu192, imeal1, jyu122, zkang7, yinzhi.cao}@jhu.edu
Johns Hopkins University

A Artifact Appendix

A.1 Abstract

In the paper, we present Hulk, the first dynamic analysis framework to automatically detect and exploit DOM Clobbering gadgets. Hulk operates in three phases. First, it performs dynamic taint analysis in the browser to track values from attacker-clobberable sources to sinks. Next, it leverages the recorded taint traces to construct *Symbolic DOM*, a formalized representation for defining and resolving DOM-related constraints in gadgets, and generates corresponding HTML payloads. Finally, Hulk injects the generated payloads into target webpages to verify their exploitability.

This artifact provides the source code, datasets, and instructions to reproduce the evaluation results, namely, detecting all zero-day gadgets in the *dom-clobbering-collection* and additional ones from the Tranco Top 5,000 websites.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact poses no security or privacy risks to artifact reviewers, testing websites, or their users. The tool identifies DOM Clobbering vulnerabilities and verifies them with non-destructive proof-of-concept exploits. On the *dom-clobbering-collection* benchmark, all tests are conducted locally, ensuring no impact on real users or live systems. During large-scale evaluations on live websites, the artifact exclusively analyzes publicly accessible client-side scripts. It does not interact with server-side interfaces, user data, or website functionalities.

A.2.2 How to access

The artifacts and the *dom-clobbering-collection* benchmark are publicly accessible at the following links. We also provide the Zenodo.org link for persistent access.

- https://github.com/jackfromeast/TheHulk
- https://github.com/jackfromeast/dom-clobbering-collection
- https://zenodo.org/records/15054169

A.2.3 Hardware dependencies

There is no specific hardware features required for this artifact evaluation. During our research, we performed the experiments on an Intel(R) Core(TM) i9-14900K CPU with 192 GB RAM and 100 GB of disk space.

A.2.4 Software dependencies

The experiments require Google Chrome, Node v18.18.2, and Python $\geq$ 3.10. In our setup, we used Google Chrome version 126.0.6478.126, installed at `/usr/bin/google-chrome` on Ubuntu 24.04.1 LTS. We recommend using the same version for consistency. Other modern versions may also work but are not guaranteed. All additional software dependencies can be installed during the Set-up step.

A.2.5 Benchmarks

To evaluate the functionality and reproducibility of this artifact, we provide the following two benchmarks:

- The *dom-clobbering-collection* benchmark. This dataset includes 28 zero-day gadgets discovered by Hulk from widely used client-side libraries. Each gadget is documented by a README file detailing the vulnerable code and a proof-of-concept (PoC).
- The large-scale benchmark. This dataset consists of 497 zero-day gadgets identified by Hulk across 354 domains from the Tranco Top 5,000 websites. This benchmark is used to evaluate Hulk's effectiveness in detecting gadgets on real-world live websites at scale.

A.3 Set-up

A.3.1 Installation

- Step 1: Clone our GitHub repository recursively using the following command: `$ git clone -recursive`

`https://github.com/jackfromeast/TheHulk.git`

- Step 2: Navigate to the project root and run the dependency installation script:

  `$ cd TheHulk && ./install.sh`

- Step 3: Configure the network proxy for HTTP(S) traffic to `http://127.0.0.1:8899`. This step is crucial as the experiment requires the browser to route its traffic through a man-in-the-middle (MITM) proxy. For detailed instructions on configuring a network proxy, refer to the Ubuntu official guide[1].

A.3.2 Basic Test

To make sure that the setup is ready and well configured, we provide a command to run Hulk against our motivating example as a basic test:

```
$ ./tasks/ae-run-basic-check/run.sh
```

After running the command, the results will be stored under directory: `tasks/ae-run-basic-check/output/domain/url-hash/crawler/`, where `domain` is `127.0.0.1`, and `url-hash` is the hash value of the tested URL. The taint analysis results are stored in `taintflows.json` and the generated exploits are stored in `exploit.txt`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We present a dynamic taint analysis tool that can detect DOM Clobbering gadgets at runtime and generate detailed taint traces. This claim is supported by Experiment (E1), where we show Hulk can detect all the gadgets in the *dom-clobbering-collection* benchmark.

(C2): We introduce a *Symbolic DOM*-based method to automatically generate exploitable HTML markups from taint flow records. This claim is supported by Experiment (E2), where we show Hulk is able to generate valid exploits of all the gadgets on the *dom-clobbering-collection* benchmark.

(C3): Our approach is capable of detecting and exploiting DOM Clobbering gadgets on live websites at scale. This claim is supported by Experiment (E3), where we show Hulk can detect and exploit all the zero-day gadgets from Tranco Top 5,000 websites.

A.4.2 Experiments

All experiments of Hulk are driven by scripts in the `task` directory. Each subdirectory under `task` corresponds to a specific task, where users can define inputs, outputs, and configurations using `.yaml` files. To reproduce the experiments, refer to the tasks prefixed with `ae-`, as these are pre-configured for artifact evaluation.

(E1): *[Gadgets Detection] [3 human-minutes + 10 compute-minutes]: Running Hulk on the dom-clobbering-collection benchmark to generate detailed taint flow records.*
Preparation: No additional preparation required. For environments with a display (check with `echo $DISPLAY`) , one can set `chrome.HEADLESS` to `false` in `config.browser.yml` under the same directory to visualize the running.
Execution: Run gadget detection on *dom-clobbering-collection* benchmark with following command:

```
$ ./tasks/ae-gadget-detection-e1/run.sh
```

Results: After the task completes, the results are available under `ae-gadget-detection-e1/output` directory. The overall summary is located at `tasks/ae-gadget-detection-e1/AE-E1-Timestamp/127.0.0.1/taintflows_summary.json`, where all 28 cases are expected to be successful. Detailed taint flows for each gadget can be found at `AE-E1-Timestamp/127.0.0.1/url_hash/crawler/taintflows.json`.

(E2): *[Exploit Generation and Verification] [3 human-minutes + 30 compute-minutes]: Running Hulk to generate and verify the exploits for the gadgets in the dom-clobbering-collection.*
Preparation: In the configuration file: `tasks/ae-exploit-generation-e2/config.browser.yml`, update the `others.EXPLOIT_INPUTS_FOLDER` with the path of the output directory of E1, which is `tasks/ae-gadget-detection-e1/AE-E1-Timestamp`.
Execution: Run exploit generation and verification with the following command:

```
$ ./tasks/ae-exploit-generation-e2/run.sh
```

Results: After the task is completed, the results are available under the `ae-gadget-detection-e2/output` directory. Each web page folder should contain two newly generated files: `exploit.txt` and `verified_exploit.txt`. The former contains the generated exploits, and the latter contains the verified exploits. Each web page is expected to have at least one verified exploit in `verified_exploit.txt`.

(E3): *[Large-scale Testing] [5 human-minutes + 7 compute-hours]: Running Hulk on the large-scale benchmark to detect the zero-day gadgets that Hulk found from the Tranco Top 5,000 websites.*
Preparation: No additional preparation required.
Execution: The large-scale test encompasses the entire pipeline of Hulk, including gadget detection, exploit generation, and verification. The execution should be conducted in the following steps:

[1] https://help.ubuntu.com/stable/ubuntu-help/net-proxy.html.en

First, run the command to detect taint flows:

`$ ./tasks/ae-gadget-detection-e3-1/run.sh`.

Next, update the `others.EXPLOIT_INPUTS_FOLDER` in `tasks/ae-gadget-detection-e3/config.browser.yml` with the path to the output directory generated in the previous step (same as E2).
Finally, run the command to generate exploits and verify the gadgets:

`$ ./tasks/ae-gadget-detection-e3-2/run.sh`.

Results: After the task completes, the results are available under the `ae-gadget-detection-e3-2/output` directory. Note that the results may differ slightly from the original results (i.e., the results we obtained between July 2024 and August 2024) due to regular updates on live websites or patches applied after our disclosure.

A.5 Notes on Reusability

The major components of Hulk—the dynamic taint analysis tool and the *Symbolic DOM*-based exploit generator—are designed to facilitate future client-side security testing. The dynamic taint analysis tool is extensible, allowing users to customize taint configurations (e.g., sources, sinks, and propagation rules). The exploit generator enables researchers to define DOM-related constraints and automatically generate corresponding HTML markups.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: FIXX: FInding eXploits from eXamples

Neil P Thimmaiah
University of Illinois Chicago
npemma2@uic.edu

Yashashvi J Dave
University of Illinois Chicago
yashashvidave@gmail.com

Rigel Gjomemo
University of Illinois Chicago
rgjome1@uic.edu

V.N. Venkatakrishnan
University of Illinois Chicago
venkat@uic.edu

A Artifact Appendix

This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of our novel approach FIXX. It includes a description containing all the software requirements that should be met before proceeding to run FIXX. This appendix also contains details on how to set up your environment to run FIXX.

A.1 Abstract

FInding eXploits from eXamples (FIXX) is a novel approach focused on detecting taint-style vulnerabilities in PHP applications. Using important information from a CVE containing details regarding a vulnerability in an application, our method aims to discover possible similar vulnerabilities in the corresponding application. This artifact contains details on how FIXX can be used to analyze web applications and detect vulnerable paths similar to a previously known vulnerable path. It contains all the requirements needed to set up the environment, as well as details on how FIXX can be installed. Finally, the major claims of the paper have been discussed, and detailed experiments on how these claims can be reproduced.

A.2 Description & Requirements

In this section, we describe the artifact used to run FIXX as well as discuss the requirements needed to run the artifact. Our approach, FIXX, can detect similar exploitable paths in a web application that are similar to a previously known vulnerable path in the corresponding application. FIXX is primarily written in Python and requires certain libraries and packages to perform its analysis. We have discussed each of these dependencies in the subsections below.

A.2.1 Security, privacy, and ethical concerns

FIXX does not pose any security or data privacy concerns when it's being used to discover exploitable paths in a given application. A minor concern: during the primary prerequisite step of FIXX's analysis, which involves reproducing the exploit wherever possible for applications to be analyzed, the evaluators should not use personal information when attempting to create accounts during the installation of applications.

A.2.2 How to access

Our artifact can be accessed using the following URL:
https://doi.org/10.5281/zenodo.14738531

A.2.3 Hardware dependencies

There are no hardware dependencies required to run FIXX.

A.2.4 Software dependencies

The artifact required to run FIXX does require certain software packages and libraries to be installed to perform the analysis. The most important thing to note here is the artifact should be run inside a docker container. To make it easier, we have provided a docker image that contains all the requirements needed to create a new docker container that supports the analysis. We have outlined all the requirements in a docker file called docker-compose.yaml as well as provided instructions on how to retrieve our docker image from Docker Hub in the subsection Set Up. We have also included all the packages in the respective files of the code so that there is no additional error while installing FIXX. Finally, in order to fine-tune the GPT4 model with your own custom key, the library openai needs to be installed.

A.2.5 Benchmarks

FIXX doesn't require any dataset to run the primary results presented in the paper. However, to obtain the precise metrics of the GPT4 model reported in our paper, the evaluators are required to create a personalized GPT4 model and fine-tune it with 50 CVE descriptions. While we have provided the model ID used to obtain the results presented in the paper, OpenAI prevents the sharing of API keys and disables them when they are uploaded to digital repositories. However, we have provided a dataset of 50 unique CVE descriptions that the evaluators can use to fine-tune their custom GPT4 model. Once that is complete, the authors can use any of the CVEs reported in Table 2 of our paper to test their model and reproduce the metrics reported in the paper.

Additionally, FIXX requires the request data as well as the malicious payload that is sent as input to every application when performing the analysis. We have provided a detailed table within our artifact that contains the corresponding files for each of the applications discussed in our paper. Lastly, since FIXX detects similar vulnerabilities to a known vulnerability, while not mandatory for all applications, the exploit must be thoroughly reproduced in order for FIXX to be able to discover accurately the other similar paths in the application. This could be inputting the malicious payload into the vulnerable parameter, inserting the payload into the database, etc. We have provided a detailed example within our artifact pertaining to this.

A.3 Set-up

The following steps detail all the requirements needed to set up your environment to run FIXX:

Docker Image : Obtain the image neildocker693/fixx from Docker Hub. The image can be pulled using the command: **docker pull neildocker693/fixx**. This image should have all the requirements required to run FIXX.

Docker Container : Once the image has been pulled, you can build the container using the command: **docker-compose up -d fixx**. This will create and start the container. You can verify that the container is running using the command: **docker ps**

Docker SSH : Finally, you can ssh into the created container using the command: **docker exec -it fixx /bash/bin**

Please note that if you encounter an error associated with the neo4j database not being available, check the corresponding database name that was loaded and make sure it matches the name of the application you are trying to process. Please refer to the README provided in the zip file of FIXX for troubleshooting such errors.

A.3.1 Installation

Once the artifact has been downloaded from the URL provided in Section A.2.2, unzip and extract its contents into the /opt/project/FIXX/ directory specified inside the docker container.

A.3.2 Basic Test

To test whether the docker container has been set up according to the requirements, you can run the command: **python fixx_main.py -n <application_name> –loadcpg**, which will load the code property graph (CPG) of the corresponding application. Note that if you are analyzing an application outside of the ones provided within our artifact, you will need to build the CPG of the application. This can be done using the command: **python fixx_main.py -n <application_name> –buildcpg**. Once the CPG of the application has been loaded, please refer to the README file within our artifact for a few intermediate steps before analyzing the applications using FIXX.

A.4 Evaluation workflow

We represent the main claims made by the paper and outline the steps needed to be followed to reproduce the results to validate those claims in the following subsections:

A.4.1 Major Claims

In this section, we provide the major claims made in our paper, along with the corresponding experiments that can done to reproduce the results of these claims.

(C1): FIXX can extract the most reused and sensitive instructions in a given PHP application, referred to as seeds, by computing their average reusability scores, sensitivity scores as well as selection scores. This is proven by the experiment (E1) in Section 5 whose results are described in column 1, 2 and 3 of Table 2 in the paper.

(C2): FIXX discovers multiple similar nodes at a specified similarity modulo for the maximum number of seeds (1–4) for each of the selected PHP applications. This is proven by the experiment (E2) in Section 5 whose results are described in column 4 of Table 3.

(C3): FIXX detects multiple similar paths (*sPaths* and *dPaths*) at the same similarity modulo value for each of the obtained similar nodes at a specified similarity threshold for the selected PHP applications. This is proven by the experiment (E3) in Section 5 whose results are described in column 5 of Table 2 as well as column 5 and column 6 of Table 3 respectively.

(C4): FIXX detects multiple new exploitable paths, 10 of which have been reported to MITRE and published as new CVEs. This is proven by the experiment (E4) in Section 5 whose results are described in column 7 and column 9 of Table 3.

(C5): FIXX can rediscover the original exploitable path described in the vulnerability of a given application by

using the newfound *sPaths* and *dPaths*. This is proven by the experiment (E5) in Section 5 whose results are described in column 8 of Table 3.

A.4.2 Experiments

In this section, we provide a detailed set of experiments that can help reproduce the results reported in the paper and support the claims made in the previous section. We also provide the running times for each of the experiments. Please note that since our approach has been evaluated on 19 unique PHP CVEs, we have provided an approximate compute time and human time for all the applications combined, which also includes a bit of manual work when switching between applications described in the CVEs. Finally, for any reported initial CVE in the paper, we recommend verifying all the claims associated with that CVE before proceeding with another CVE to save time and effort.

(E1): [Seed Scores] [*2 computed hours + 1 human hour*]: This experiment will aim to obtain the maximum number of seeds (1–4) for each of the applications as well as compute their reusability, sensitivity and selection scores as described in Table 2. By reproducing these results, the claim C1 can be verified.

How to: To run this experiment, you will need the command **python fixx_main.py -n <application_name> –seedanalysis -vulnerablefile <vulnerable_file_name> -seeds 4**. Note that, as mentioned in the paper, certain applications have limited seeds, so you will need to reduce the seeds for a few of them accordingly. The maximum number of seeds for each of the applications can be found in column 5 of Table 2.

Preparation: To prepare FIXX to perform this analysis for a given application, we need to load the corresponding Code Property Graph (CPG) of the application in the docker container. This can be done using the command **python fixx_main.py -n <application_name> –loadcpg**. Next, make sure to obtain the execution trace of the vulnerable file according to the CVE description of that application. Please refer to the provided README on how to gather the execution trace before analyzing an application. Once the CPG has been loaded and the execution trace has been collected, you can proceed to execute the experiment.

Execution: Run the command provided in the **How to**, which will compute the total number of seeds, i.e., 4, which was used to evaluate the applications in our paper, as well the average reusability scores, sensitivity scores and selection scores for each of them. Note that you are free to evaluate any of the applications on any number of seeds. The higher the number of seeds, the more instructions, although less sensitive, will be considered when computing similar paths

Results: Once FIXX has extracted the seeds and computed the corresponding scores for them, the results will be stored in the /results/seed_results.txt file. This file will contain the location for each of the 4 requested seeds in the corresponding application, as well as their reusability, sensitivity, and selection scores. Finally, the average scores can be found at the end of the file, which can be compared to the results in columns 1, 2, and 3 of Table 2 in our paper.

(E2): [Similar Nodes Detection] [*2 computed hours + 1 human-hour*]: This experiment will aim to obtain unique, similar instructions at a specified similarity modulo for each of the detected seeds.

How to: To run this experiment, you will need the command **python fixx_main.py -n <application_name> –computesimilarnodes -vulnerablefile <vulnerable_file_name> -seeds 4 -modulo 2**.

Preparation: To evaluate our claim, a modulo of 2 can be used. However, FIXX accepts any positive modulo, including 0.

Execution: Run the command provided in the **How to**, which will compute the total number of unique, similar instructions at a modulo 2 for each of the 1–4 seeds, depending on the application, which was used to evaluate the applications in our paper. Note that you are free to evaluate any of the applications on any modulo. The higher the modulo, the lesser the similarity between instructions being considered by FIXX

Results: The sim_nodes_results.txt file inside the results folder can be viewed to verify the different similar instructions along wth their location in the application that have been detected by FIXX. These results correspond to column 4 in Table 3 of our paper.

(E3): [Path Detection] [*15 compute-hour + 10-12 human-hours*]: This experiment will aim to obtain the *sPaths* and *dPaths* reported in column 5 of Table 2 and column 5, 6 of Table 3.

How to: To run this experiment, you will need the command **python fixx_main.py -n <application_name> –detectsimilarpaths -vulnerablefile <vulnerable_file_name> -seeds 4 -modulo 2 -threshold 50**.

Preparation: To evaluate our claim, a threshold value of 50 must be used. However, FIXX accepts any threshold values between 0 and 100 (both included)

Execution: Run the command provided in the **How to**, which will compute the total number of *sPaths* and *dPaths* at a similarity modulo of 2 and a threshold of 50 for each of the 1–4 seeds and detected similar instructions, depending on the application, which was used to evaluate the applications in our paper. Note that this step can take a while, depending on the size of the application. For the extremely large applications SeoPanel and Collabtive, we have written additional code in the fixx_similar_exploits_analyzer.py file. Just search for

SeoPanel or Collabtive, depending on the application that you are testing, and uncomment the corresponding block of code. This will make the processing time of FIXX even faster due to optimal cypher queries that have been written for these applications.

Results: The *sPaths* and *dPaths* per seed can be verified from the terminal and correspond to column 5 of Table 2 in the paper. The path_results.txt file inside the results folder can be viewed to verify the *sPaths* and *dPaths* that have been detected by FIXX. This file contains each of the complete paths along with the total number of *sPaths* and *dPaths* which correspond to column 5 and column 6 in Table 3 of our paper.

(E4): [Exploitable Paths] [*10 human hours*]: Note that this experiment is based on the results obtained from the previous experiment. E3 returns a list of the *sPaths* and *dPaths*. It also prints the total number of exploitable paths in the same text file. However, as mentioned in the paper, in addition to the paths detected by FIXX, we verify the exploitable paths manually. So, we check each of the paths returned by FIXX in E3 and verify the total number of exploitable paths, which correspond to Column 7 of Table 3 in our paper. An example of this is shown in the Example README file.

(E5): [Re-Discover CVE] [*20 compute-hour + 10-12 human-hours*]: For this experiment, we have listed the running time for all the applications combined. This experiment will aim to obtain the original exploitable path reported in the CVE using the newfound *sPaths* and *dPaths*. Note that this experiment can take a lot of time as it repeats the entire analysis of FIXX, and we only report the result as a Yes or No in column 8 of Table 3 in the paper, which verifies if the original path was found or not. We have provided a thorough example in our Example.md file

How to: To run this experiment, you will need the command **python fixx_main.py -n <application_name> –detectsimilarpaths -vulnerablefile <vulnerable_file_name> -seeds 4 -modulo 2 -threshold 50**. Please note that the value of the threshold will need to be altered accordingly since certain paths cannot be rediscovered with the same threshold due to limitations in terms of ancestor/descendant nodes or data flow paths, as discussed in our paper.

Preparation: To evaluate this claim, you will need to select any one of the obtained *sPaths* and *dPaths* and obtain the exploit string as well as the request_data.json file.

Execution: Before running the command provided in the **How to**, you will need to obtain the execution trace of the file in which the selected *sPath* or *dPath* is present. Once the trace has been obtained, you can run the above command to repeat the analysis of FIXX. The path results will be stored in the same folder and can be verified to check if the original vulnerable path is present or not.

Results: The path_results.txt file inside the results folder can be viewed to verify the new *sPaths* and *dPaths* that have been detected by FIXX. You can manually verify that the original exploitable path is present among these paths and confirm with the results shown in column 8 of Table 3.

A.5 Notes on Reusability

Our artifact can be used to detect all exploitable paths in any PHP web application similar to a previously known vulnerable path present in that application. You can modify the values of the seeds, modulo as well as the threshold parameter to discover different types of exploitable paths. Please refer to our paper to avoid resulting in false positive cases by increasing the value of the seeds and similarity modulo or decreasing the value of the similarity threshold parameters, respectively. Significantly increasing the seed value or modulo value can lead to FIXX selecting instructions that are not quite sensitive or differ quite a bit in terms of the identifiers involved. Similarly, decreasing the threshold significantly can lead to FIXX selecting paths that are not similar to the original path. Although FIXX has been evaluated on vulnerabilities like Cross-Site Scripting and SQL Injection, the artifact can discover paths associated with other vulnerability types like Command Injection, etc., as long as the right request is sent when obtaining the execution trace, as this is the basis for discovering similar instructions that lead to similar paths.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: Effective Directed Fuzzing with Hierarchical Scheduling for Web Vulnerability Detection

Zihan Lin, Yuan Zhang, Jiarun Dai, Xinyou Huang, Bocheng Xiang, Guangliang Yang, Letian Yuan, Lei Zhang, Fengyu Liu, Tian Chen, and Min Yang

Fudan University

A Artifact Appendix

A.1 Abstract

This is the artifact evaluation appendix for paper "Effective Directed Fuzzing with Hierarchical Scheduling for Web Vulnerability Detection." In this work, we propose a novel directed fuzzing approach, called WDFuzz, that can effectively vet the security of Java web applications. Our WDFuzz approach is two-fold. First, we develop a semantic constraint extraction technique to accurately capture the expected input structures and constraints of web parameters. Second, we implement a hierarchical scheduling strategy that evaluates the potential of each seed to trigger vulnerabilities and prioritizes the most promising seeds. In our evaluation against real-world Java web applications, WDFuzz achieved a 92.6% recall rate in the known vulnerability dataset, finding 3.2 times more vulnerabilities and detecting them 7.1 times faster than the state-of-the-art web fuzzer Witcher.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The evaluation process does not raise any ethical concerns. Our benchmark dataset consists of older versions of web applications with known vulnerabilities. When deploying these applications on the evaluator's local machines, it is recommended not to allow external access to these applications to prevent attacks from external attackers.

A.2.2 How to access

The artifact can be downloaded from the Zenodo record[1]. After downloaded, first concatenate all the splitted compressed files with the following command:

```
cat WDFUZZ_part_* > WDFUZZ.tar.gz
```

Then unzip the file with this command:

```
tar xvf WDFUZZ.tar.gz
```

[1] https://zenodo.org/records/15128608

A.2.3 Hardware dependencies

At least 60 GB RAM and 200 GB free disk space are needed to run the Artifact Evalution. We run our experiments on a Ubuntu 18.04 server with 64-core Intel E7-4820 v2 2.00GHz CPU, 173 GB RAM and 1 TB disk.

A.2.4 Software dependencies

A Unix-like system is required for Artifact Evaluation. We recommend using Ubuntu 18.04. The other software dependencies are listed below.

- **Static Analysis**. Java 17[2] is required to run the Tai-e static analysis.
- **Instrumentation**. (1) Java 8[3] and (2) mvn[4] are needed to compile the instrumentation program. (3) GCC[5] is needed to compile the `LD_PRELOAD` binary.
- **Dynamic Fuzzing**. (1) Python3[6] and libraries (i.e., requests, pyyaml, and selenium) is needed to run the fuzzer's wrapper scripts. (2) Rust[7] is needed to run the core `LibAFL` fuzzer. (3) Docker[8] is needed to set up Web applications. (4) Chrome and chrome-driver[9] are needed to login web applications automatically.

A.2.5 Benchmarks

Our artifact contains 12 open-source web applications as benchmark dataset. Please refer to Table 1 in the paper for the detailed application list. The Docker images of these web applications are located in the `./dataset` directory within the artifact. To implement and setup these

[2] https://www.oracle.com/java/technologies/javase/jdk17-0-13-later-archive-downloads.html

[3] https://www.oracle.com/java/technologies/javase/javase8u211-later-archive-downloads.html

[4] https://maven.apache.org/download.cgi

[5] Use `sudo apt install build-essential` to install.

[6] Use `sudo apt install python3 python3-pip` to install.

[7] https://forge.rust-lang.org/infra/other-installation-methods.html

[8] https://docs.docker.com/engine/install/ubuntu/

[9] https://googlechromelabs.github.io/chrome-for-testing

applications for fuzzing, please refer to the document in `quick-reproduction-guide.pdf`. Due to legal and licensing reasons, we cannot provide the 3 closed-source commercial web applications tested in the paper for Artifact Evaluation. We believe that the experiment results from the open-source applications are sufficient to prove the claims in the paper.

A.3 Set-up

A.3.1 Installation

Python3, Rust, chrome and chrome-driver must be installed to run WDFuzz.

Python3 and library installation. First, we recommend using this command on Ubuntu servers to install Python3 environment:

```
sudo apt install python3 python3-pip
```

Then use this command to install necessary Python3 libraries:

```
pip install requests pyyaml selenium
```

Rust, chrome and chrome-driver installation. We recommend installing Rust, chrome and chrome-driver according to the instructions on their official websites, which are listed in the footnote.

Web applications installation. Before start fuzzing, web applications under test should be installed. We recommend to set up these web applications using the Docker images in the artifact (within `./dataset` directory) and following the guidance in `quick-reproduction-guide.pdf`. The baseline tool Witcher is also installed in the Docker images.

[Optional] Java, GCC, and mvn installation. Java, GCC, and mvn are *optional* to install because we have pre-compiled binaries for instrumentation and ready-to-use static analysis results. To recompile these components, please refer to the installation guide in `WDFuzz-README.pdf`.

A.3.2 Basic Test

Please run the following commands to check if necessary environments have been installed. The output examples are the outputs obtained in our test environment.

- **Check Rust environment.**
  Command:
  ```
  cargo --version
  ```
  Output example:
  ```
  cargo 1.79.0 (ffa9cf99a 2024-06-03)
  ```
  Command:
  ```
  rustc --version
  ```
  Output example:
  ```
  rustc 1.79.0 (129f3b996 2024-06-10)
  ```
- **Check Python3 environment.**
  Command:
  ```
  python3 --version
  ```
  Output example:
  ```
  Python 3.6.9
  ```
- **Check Docker environment.**
  Command:
  ```
  docker --version
  ```
  Output example:
  ```
  Docker version 20.10.21, build
  20.10.21-0ubuntu1~18.04.3
  ```
- **Check chrome-driver environment.**
  Command:
  ```
  chromedriver --version
  ```
  Output example:
  ```
  ChromeDriver 132.0.6834.83 ...
  ```
- **Check web applications under test.**
  You just need to use your browser to check if the websites can be successfully opened.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): WDFuzz achieves 92.6% recall in known vulnerability detection of web applications. Compared to the state-of-the-art web application fuzzer Witcher, WDFuzz finds 3.2 times more vulnerabilities and detects them 7.1 times faster. This is proven by the experiment (E1) described in §5.3 whose results are reported in Table 1.

(C2): Each module of WDFuzz (i.e., entry extraction, constraint extraction, and hierarchical scheduling) contributes to a notable increase in recall rates for known vulnerabilities, with improvements ranging from 20% to 30%. This is proven by the experiment (E2) described in §5.5 whose results are reported in Table 2.

A.4.2 Experiments

(E1): Known vulnerability reproduction [10 human-hours + 40 compute-hours + 100GB disk]: this experiment demonstrates that the proposed WDFuzz outperforms the state-of-the-art fuzzer Witcher in both vulnerability detection effectiveness and efficiency by reproducing the known vulnerabilities in the benchmark web applications.

Preparation: First, please make sure the environments being set up as described in §A.3.1, including Python3, Rust, Docker, chrome, chrome-driver, and web applications under test. Then some directories must be created before testing by running commands `mkdir $HOME/wdfuzz_data`, `mkdir $HOME/wdfuzz_data/fuzz_apps_class` and `mkdir $HOME/wdfuzz_data/jdf_data`.

Execution: Run `fuzz_<app>.sh` in `./DF` directory for the full experiment of WDFuzz, and run

`fuzz.sh` in the docker container of each web application for the Witcher experiment. You can refer to `quick-reproduction-guide.pdf` for more details.

Results: The vulnerability detection results of WDFuzz will be in the directory `$HOME/wdfuzz_data/<date>_<app>`. Each record in the text file `vuln_testcase.log` means that WDFuzz has detected a vulnerability. Please refer to the entry paths of the known vulnerabilities in Table 1 to get the recall rate of the known vulnerabilities. Vulnerabilities that are not in Table 1 are the unknown vulnerabilities detected by WDFuzz.

The results of Witcher experiment will be inside the docker container. Typically, the results are under the directory `/root/user/WICHR/<app>-WICHR`. Please check the `run_summary.txt` file to get the vulnerabilities identified by Witcher.

We provide a set of utility scripts within `log_comparison_scripts.zip`. For comparing WDFuzz's results, you can use the following command to extract the recall of known vulnerabilities from the logs: `python3 wdfuzz_log_compare.py --project <web app name> --log <path to vuln_testcase.log>`. For comparing Witcher's results, use the following command: `python3 witcher_log_compare.py --project <web app name> --log <path to run_summary.txt>`.

(E2): Ablation study [15 human-hours + 60 compute-hours + 30GB disk]: this experiment proves that each module of WDFuzz makes a notable contribution to vulnerability detection by evaluating the performance after ablating each module of WDFuzz.

Preparation: The same as the experiment (E1).

Execution: Run `fuzz_<app>_ablation<1 or 2 or 3>.sh` in `./DF` directory for the ablation study. Please note that for some web applications, CrawlerGo failed to crawl any usable URLs, and as a result there is no `ablation_1` experiment (i.e., the WDFuzz$_b$ baseline).

Results: The same as the experiment (E1).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

Table 1: Known vulnerablilities.

Application	Vulnerable entry path

jeecg-boot	`/sys/duplicate/check`
	`/jmreport/queryFieldBySql`
	`/jmreport/show`
	`/sys/api/queryTableDictItemsByCode`
	`/sys/dict/queryTableData`

jeesite	`/a/act/task/start`
	`/userfiles`
	`/a/sys/user/export`

jshERP	`/user/list`
	`/msg/getMsgCountByStatus`
	`/msg/list`
	`/unit/list`
	`/role/list`
	`/account/list`
	`/depotHead/list`
	`/materialProperty/list`

MCMS	`/ms/mdiy/dict/list` (CVE-2021-46383)
	`/ms/cms/category/list`
	`/ms/mdiy/page/verify`
	`/ms/template/writeFileContent`
	`/mdiy/dict/list`
	`/ms/mdiy/dict/list` (CVE-2022-27466)
	`/cms/content/list` (CVE-2022-26585)
	`/ms/template/writeFileContent` (CVE-2021-46062)
	`/ms/template/writeFileContent` (CVE-2021-46063)
	`/cms/content/list` (CVE-2022-23898)
	`/ms/template/unZip`
	`/mdiy/dict/listExcludeApp`

RuoYi	`/system/dept/edit`
	`/demo/form/localrefresh/task`
	`/common/download/resource`
	`/common/download`
	`/system/role/authUser/unallocatedList`
	`/system/role/export`
	`/system/role/authUser/allocatedList`
	`/system/user/export`
	`/system/role/list`
	`/system/user/list`
	`/system/dept/treeData`

SpringBlade	`/blade-user/export-user`
Halo	`/api/admin/themes/fetching`
DreamerCMS	`/admin/search/doSearch` `/search`

PublicCMS	`/admin/ueditor?action=catchimage` (CVE-2020-20914)
	`/admin/sysSite/execSql`
	`/admin/ueditor?action=catchimage` (CVE-2021-27693)
	`/admin/#site_sysSite/script`
	`/admin/ueditor?action=catchimage` (CVE-2024-40543)
	`/admin/cmsTemplate/replace`

Yudao	No known vulnerabilities
lamp-boot	No known vulnerabilities

WebGoat	`/SqlInjection/attack8`
	`/SqlInjection/attack4`
	`/SqlInjection/attack2`
	`/SqlInjection/attack5`
	`/SqlInjectionAdvanced/challenge`
	`/SqlInjection/attack10`
	`/challenge/5`
	`/SqlInjection/attack3`
	`/SqlInjection/attack9`
	`/SqlInjection/assignment5a`
	`/SqlInjectionAdvanced/attack6a`
	`/SqlInjectionMitigations/servers`





USENIX Security '25 Artifact Appendix: Leuvenshtein: Efficient FHE-based Edit Distance Computation with Single Bootstrap per Cell

Wouter Legiest, Jan-Pieter D'Anvers, Bojan Spasicm Nam-Luc Tran, Ingrid Verbauwhede

A Artifact Appendix

The following is an Artifact Appendix for the USENIX Security '25 conference, detailing the Leuvenshtein algorithm for efficient Fully Homomorphic Encryption (FHE)-based edit distance computation. This document provides a roadmap for evaluating the artifact, including hardware, software, and configuration requirements, as well as instructions for reproducing claims.

USENIX Security '25 Artifact Appendix: Leuvenshtein: Efficient FHE-based Edit Distance Computation with Single Bootstrap per Cell

A.1 Abstract

Our paper develops a new algorithm to efficiently calculate the edit distance on encrypted data using Fully Homomorphic Encryption. The artifact is a Rust project that implements the Leuvenshtein algorithm. The code is available via a GitHub link and a Zenodo repository. Both repositories are equivalent, but the GitHub repository is easier to clone due to its folder structure.

A.2 Description & Requirements

This section provides the necessary information to recreate the experimental setup used for this artifact. It includes the minimal hardware and software requirements , and details benchmarks used to produce the results.

A.2.1 Security, Privacy, and Ethical Concerns

There are no security, privacy, or ethical concerns as the artifact evaluation involves running a Rust or Python program.

A.2.2 How to Access

Our codebase can be accessed at: https://github.com/WoutLegiest/leuvenshtein_ae. For completeness, the code is also available at: https://zenodo.org/records/15871491. However, we recommend using the GitHub repository due to its support for a folder structure.

A.2.3 Hardware Dependencies

Our experiments were conducted on a dual AMD EPYC 9174F 16-Core Processor (64 threads in total) running Ubuntu 22.04, with 512 GiB RAM. As we are aiming only for the functional badge, the exact setup and outcomes are not critical.

A.2.4 Software Dependencies

Python: Versions 3.8 to 3.12. Rust: Version 1.80.

A.2.5 Benchmarks

Within the main file, example strings of lengths 8, 100, or 256 can be selected for testing.

A.3 Set-up

Our experiments were run using rustup version 1.80.

A.3.1 Installation

Rust Our experiments were conducted using Rust version 1.80. To install this specific Rust version, use `rustup install 1.80.0`. Further instructions can be found at https://www.rust-lang.org/tools/install. Rust can be installed using the following command:

Can be installed through:

- `curl -proto '=https' -tlsv1.2 -sSf https://sh.rustup.rs | sh`
- After installation, reload the terminal or source the Rust environment file as indicated in the final step of the installation. Then, install the specific Rust version:
- `rustup install 1.80.0`

The remaining dependencies will be downloaded by the Cargo tool.

Concrete The concrete library is a python library that needs to be installed with

1. Install the concrete library: `pip install concrete-python==2.8.1`
2. Clone the correct Concrete version `git clone https://github.com/zama-ai/concrete ; cd concrete ; git reset --hard fd9db128869818293d3b4336f44e5938cfe5c480`
3. Copy the patch file to the folder and apply our patch to the repo: `git apply ../concrete_ascii.patch`

A.3.2 Basic Test

The basis implementation can be run with

- `cargo +1.80 run -release`

Our preprocessing version can be run with:

- `cargo +1.80 run -release -bin leuvenshtein_preprocess`

This will run the algorithm and the plaintext algorithm and will time the encrypted execution.

Concrete

- Go to Levenshtein distance folder: `cd concrete/frontends/concrete-python/examples/levenshtein_distance/`
- Run the wanted size of concrete instance
  - `python3 levenshtein_distance.py --distance abcdefgh adcdefgf --alphabet ascii --max_string_length 8`
  - `python3 levenshtein_distance.py --distance BEgfEHGfShHtvKazXNeEvNWmvfbrAWyAYZjkXvNkmEajQNCTKZnkPeEDadvQtUnGhJRpWZASUMfXArGZFSUYgFeCAWxSvKNdpsnV EEhjZMnFsMFCsKnnyZvtrPeKxfmJfJVJNcYAwYmrGNgTUUSAgduNQZttWFdYFKddcKjkUEpPUmGkszZSVVNWkThxSFRgMzrbqATe --alphabet ascii --max_string_length 100`
  - `python3 levenshtein_distance.py --distance mizf30WE1Pzqu0HtZKMMCE6f3NE2TGPPYmgSunfkBJqGJqveg97i61fu5KG7z8UmR5DVVALk5CCL0fzEv687LdRuunZ8SYUFmQEf66dXZ6vejGKR7HhSiY5XLWbCYFddqtFEX2QjmHRmqB7tngfG7m0CBX1D6wcvLyYp0rpiv1GSw5T1ZfuT2mcjHUi05zd6N4EqLmTP8ETEc2vQaJcVGSaXRETejNebwwb4m9wUUMd0abrEQeLw3Ubn9un6tTeP yVL0hi1V8mUQakFWZGEHwQBduvaTtKC6dNMbnyEiU29pKdwLfCnY7jWceXQizTiwhGxiAkuwvdcgpTtaZW3XJ6t75HL86nV2QPUWaxipf6x7JE8NPQT0RrzcheaydjLdPyUhAQ3UhJ2bLVE5wtNDAgdBgX3N5Ru4iXqbFXWD5ZAbzniVaWr5iE0wQenwt8QjqERf6A67P9rLkmGPKS8LHJxttCKRBWM3qr1F93JZtEhGKcQ1079pUXcCgAvLhCWi --alphabet ascii --max_string_length 256`

A.4 Evaluation workflow

The basic workflow for processing two strings of length 8 is described in Section A.3.2. For larger string sizes, the main function needs to be adapted. In both (`main.rs` and `main_prepros.rs`) files, within the `fn main()` function, alternative inputs for ASCII encoding of lengths 100 and 256 are provided in comments. These must be uncommented to test them.

A.4.1 Major Claims

(C1): The outcome of the Leuvenshtein algorithm is the correct edit distance, calculated on encrypted data.
(C2): The execution of the Leuvenshtein algorithm is remarkably faster than the execution of the Concrete Compiler.

A.4.2 Experiments

Run the experiments as described in Section A.3.2 for string lengths 8, 100, and 256. Note that the 256-length input for Concrete is expected to take over a week to run and should not be completed. We anticipate that the decrypted outcome of Leuvenshtein will be identical to the outcome of the plaintext version and the Concrete implementation.

A.5 Notes on Reusability

The algorithm itself can be implemented in any industry context, provided the cryptographic scheme and parameters remain consistent.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Engorgio: An Arbitrary-Precision Unbounded-Size Hybrid Encrypted Database via Quantized Fully Homomorphic Encryption

Song Bian[1], Haowen Pan[1], Jiaqi Hu[1], Zhou Zhang[1], Yunhao Fu[1], Jiafeng Hua[2], Yi Chen[3], Bo Zhang[3], Yier Jin[4], Jin Dong[3], and Zhenyu Guan[1]*

[1]Beihang University, [2]Huawei Technology
[3]Beijing Academy of Blockchain and Edge Computing
[4]University of Science and Technology of China

A Artifact Appendix

A.1 Abstract

Engorgio is a hybrid encrypted database based on quantized fully homomorphic encryption. Engorgio can efficiently evaluate fast and arbitrary-precision homomorphic filtering, sorting and complex aggregation algorithms that enable a variety of SQL queries to be applied over FHE ciphertexts. This artifact allows users to evaluate the availability and runtime latency of Engorgio across various benchmarks.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The evaluation of our artifact does not introduce any risks to machine security, data privacy, or ethical concerns.

A.2.2 How to access

Our implementation is available on Zenodo with the link https://zenodo.org/records/14730651.

A.2.3 Hardware dependencies

The experimental results detailed in the manuscript were derived from evaluations performed using an Intel Xeon Gold 6226R processor with 512 GB of RAM in a single-thread environment.

A.2.4 Software dependencies

To evaluate the artifact, the specific OS and software packages required include Ubuntu 20.04 as the operating system, along with essential development tools like build-essential, g++-10, apt-utils, ca-certificates, as well as version control (git), build tools (cmake>=3.16), and cryptographic library (libgmp-dev, libntl-dev).

A.2.5 Benchmarks

SIFT dataset.

A.3 Set-up

A.3.1 Installation

Engorgio can be built (out-of-source) by executing following commands:

```
$ cd Engorgio
$ mkdir build
$ cd build
$ cmake ..
$ make
```

A.3.2 Basic Test

Following the instructions above, nine output binaries will be generated in the `build/bin/` directory. You can run these binaries by:

```
$./bin/comparison_test
$./bin/sort_test
$./bin/sync_test
$./bin/topk_test
$./bin/relational_query_test
$./bin/vectorized_query_test_1
$./bin/vectorized_query_test_2
$./bin/hybrid_query_test_1
$./bin/hybrid_query_test_2
```

*Corresponding author.

A.4 Evaluation workflow

A.4.1 Major Claims

We enumerate the major claims of Engorgio as follows:

(C1): Engorgio achieves 28×–854× faster homomorphic comparison, 65×–687× faster homomorphic sorting, 42×–4,595× faster homomorphic synchronization, and 4×–58× faster homomorphic Top-k over the state-of-the-art (SOTA) solutions. This is proven by experiments (E1, E2, E3 and E4) whose results are illustrated in Figure 5.

(C2): Engorgio is capable of evaluating efficient relational, vectorized, and hybrid query benchmarks. This is proven by experiments (E5, E6 and E7), whose results are illustrated in Figure 7.

A.4.2 Experiments

(E1): [Homomorphic Comparison] [3 human-minutes + 30 compute-minutes]: In this experiment, we perform the homomorphic comparison operator in Engorgio, which is the basic operator for a hybrid encrypted database.
Preparation: Following the instructions in Appendix A.3.1.
Execution: Running the binary file in the `build/bin/` directory by executing `./bin/comparision_test`.
Results: The runtime latency for the 8-bit precision homomorphic comparison operators (both relational operator and equality operator) is less than 1 ms, while the latency for the 16-bit precision is around 1.5 ms, for the 32-bit precision, it is approximately 3 ms, and for the highest precision of 64 bits, the latency reaches approximately 8 ms. These evaluations were performed using one thread and produced accurate results as shown in Figure 5(a) and Figure 5(b).

(E2): [Homomorphic Sorting] [3 human-minutes + 25 compute-hours]: In this experiment, we perform the homomorphic sorting operator in Engorgio, which is the basic operator for data ordering and vector search.
Preparation: Following the instructions in Appendix A.3.1.
Execution: Running the binary file in the `build/bin/` directory by executing `./bin/sort_test`.
Results: The runtime latency for the 8-input homomorphic sorting is around 4.8×10^2 ms, while the latency for the 64-input homomorphic sorting is around 1.44×10^4 ms, for the 1024-input homomorphic sorting is around 5.74×10^5 ms, and for the 32768-input homomorphic sorting, the latency reaches approximately 3.79×10^7 ms. These evaluations were performed using one thread and produced accurate results as shown in Figure 5(c).

(E3): [Homomorphic Synchronization] [3 human-minutes + 3 compute-hours]: In this experiment, we perform the homomorphic synchronization operator in Engorgio.
Preparation: Following the instructions in Appendix A.3.1.
Execution: Running the binary file in the `build/bin/` directory by executing `./bin/sync_test`.
Results: The runtime latency for the 8-input homomorphic synchronization is around 2 ms, while the latency for the 64-input homomorphic sorting is around 95 ms, for the 1024-input homomorphic sorting is around 1×10^4 ms, and for the 8192-input homomorphic sorting, the latency reaches approximately 1.42×10^5 ms. These evaluations were performed using one thread and produced accurate results as shown in Figure 5(d).

(E4): [Homomorphic Topk] [3 human-minutes + 12 compute-hours]: In this experiment, we perform the homomorphic synchronization operator in Engorgio.
Preparation: Following the instructions in Appendix A.3.1.
Execution: Running the binary file in the `build/bin/` directory by executing `./bin/topk_test`.
Results: The runtime latency for the Top-1 in 256-input is around 1.6×10^4 ms, while the Top-4 in 256-input is around 4.2×10^4 ms, for the Top-8 in 256-input is around 5.2×10^4 ms, and for the Top-16 in 256-input, the latency reaches approximately 6×10^4 ms. These evaluations were performed using one thread and produced accurate results as shown in Figure 5(e).

(E5): [Relational Query] [3 human-minutes + 1.5 compute-hours]: In this experiment, we perform the end-to-end relational SQL query in Engorgio.
Preparation: Following the instructions in Appendix A.3.1.
Execution: Running the binary file in the `build/bin/` directory by executing `./bin/relational_query_test`.
Results: The runtime latency for executing TPC-H Q1 and TPC-H Q12 on an encrypted database with varying row sizes ($2^{11}, 2^{12}, 2^{13}$, and 2^{14}) is as follows: TPC-H Q1 takes approximately 32 seconds, 66 seconds, 134 seconds, and 270 seconds, respectively. TPC-H Q12 takes approximately 28.5 seconds, 56.8 seconds, 114 seconds, and 229 seconds, respectively. These evaluations were performed using one thread and produced accurate results as shown in Figure 7(a) and Figure 7(b).

(E6): [Vectorized Query] [3 human-minutes + 30 compute-hours]: In this experiment, we perform the end-to-end vectorized query in Engorgio.
Preparation: Following the instructions in Appendix A.3.1.
Execution: Running the binary file in the `build/bin/` directory by executing:
`./bin/vectorized_query_test_1`
`./bin/vectorized_query_test_2`.
Results: The runtime latency for executing vectorized query VQ1 and VQ2 on an encrypted database with

varying row sizes ($2^7, 2^8, 2^9$, and 2^{10}) is as follows: VQ1 takes approximately 20 seconds, 48 seconds, 113 seconds, and 266 seconds, respectively. VQ2 takes approximately 23 seconds, 64 seconds, 150 seconds, and 340 seconds, respectively. These evaluations were performed using one thread and produced accurate results as shown in Figure 7(c) and Figure 7(d).

(E7): [Hybird Query] [3 human-minutes + 30 compute-hours]: In this experiment, we perform the hybrid query experiment in Engorgio.

Preparation: Following the instructions in Appendix A.3.1.

Execution: Running the binary file in the `build/bin/` directory by executing:
`./bin/hybrid_query_test_1`
`./bin/hybrid_query_test_2`.

Results: The runtime latency for executing hybrid query HQ1 and HQ2 on an encrypted database with varying row sizes (2^6, 2^8, 2^{10}, and 2^{12}) is as follows: HQ1 takes approximately 15.3 seconds, 105 seconds, 600 seconds, and 3313 seconds, respectively. HQ2 takes approximately 9.3 seconds, 50 seconds, 259 seconds, and 1270 seconds, respectively. These evaluations were performed using one thread and produced accurate results as shown in Figure 7(e).

A.5 Notes on Reusability

None

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.



USENIX Security '25 Artifact Appendix: Qelect: Lattice-based Single Secret Leader Election Made Practical

Yunhao Wang
Yale University

Fan Zhang
Yale University

A Artifact Appendix

A.1 Abstract

Our artifact is a C++ library implementing the main single secret leader election protocol Qelect in [2]. Our main results aimed to be produced from this artifact are 1) the local computation time with party size in $\{32, 64, 128, 256, 512, 1024, 2048\}$, and 2) the communication time of 128 parties under both LAN and WAN settings. Our main claim in [2] (which is the same as in the submission version) uses the party size $G = 128$ as in section 7.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact should not pose any risk to the evaluators regarding security, or data privacy. None does our artifact has any ethical concerns. The testing data, key pairs and communications are all simulations with no real impact. Despite all these, we still recommend the evaluators start a fresh AWS EC2 instance without testing our code using their own machines.

A.2.2 How to access

Our code is public on Zenodo, which is also the version in our final submission.

A.2.3 Hardware dependencies

Our main runtime results should be reproducible on AWS EC2 c6i.2xlarge instances with 8 vCPU, 16GB RAM and Intel Xeon Scalable processors. Note that we do not make an explicit assumption on the network latency, but estimate the communication time based on the EC2 network directly. Corresponding latency figures are also documented in section 7 of our final submission (same as in the full version [2]).

A.2.4 Software dependencies

On a standard AWS EC2 c6i.2xlarge instance, we run the benchmarks with boot disk configured with Ubuntu 24.04 LTS operating system.

We also rely on the following software and libraries:

- C++ build environment
- CMake build infrastructure
- Docker environment to run our simulation
- SEAL library 4.1 (or latest version) and all its dependencies
- PALISADE library release v1.11.9 and all its dependencies

A detailed installation script is provided in the README.md file in our artifact. Again, all secrets and data are simulated, and thus no third-party models/datasets are used.

A.2.5 Benchmarks

We benchmarked our main SSLE protocol Qelect presented in section 6 in [2] (and as in our final submission version). The party size we provided to be tested for evaluators in the docker is $G \in \{32, 64, 128, 256, 512, 1024, 2048\}$, which covers our main claim w.r.t. party size $G = 128$ as in section 7.

A.3 Set-up

A.3.1 Installation

The installations and tests could be broken into two parts. One for *local computation* and the other for *communication simulation* between multiple parties. To recover the runtime of our major claim w.r.t. the party size $G = 128$, we first show how to launch such 128 AWS instances with proper configurations. Notice that later we would also simulate the local computation for $G \in \{32, 64, 128, 256, 512, 1024, 2048\}$, while the communication time will be estimated only for $G = 128$ under both LAN and WAN settings.

Instance configuration

1. In AWS console, under the region AWS us-east-2 (Ohio), create key pair named ssle_us_east, which will automatically get its pem file (rename to ssle.pem) downloaded to local.

2. In AWS console, under other three AWS regions: us-west-1 (California), eu-west-1 (Ireland), and ap-southeast-1 (Singapore), import the previously downloaded ssle.pem file to the corresponding ssle_us_west, ssle_eu_west, ssle_ap_southeast (so that the key pairs under all four regions are actually the same). This step is to guarantee that the evaluators could be able to control all instances simultaneously, in order to simulate the broadcast in between.

3. In AWS console, under four AWS regions: AWS us-east-2 (Ohio), us-west-1 (California), eu-west1 (Ireland), and ap-southeast-1 (Singapore), create a launch template with the following configuration:

   - Application and OS Images: ubuntu 24.04, x86_64
   - Instance type: c6i.2xlarge
   - Key pair: select the corresponding ssle_[region] key pair just created
   - Network setting: create security group and corresponding VPC that allows for any IPv4 address in both the inbound and outbound rules.

4. In AWS console, launch 128 instances under AWS us-east-2 (Ohio) (this will be used to simulate the communication under LAN); launch 32 instances under all other three regions respectively. In generally, with G, uniformly distribute the instances across all four regions to simulate the communication under WAN setting.

Local computation setup Ssh to one of the instances under AWS us-east-2 (Ohio) Note that the following installation only needs to be performed on a single AWS EC2 instance.

```
# install docker
sudo apt-get update
sudo apt-get install -y ca-certificates curl
sudo curl -fsSL https://download.docker.com \
/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

echo \
  "deb [arch=$(dpkg --print-architecture)
signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo
"$UBUNTU_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list
> /dev/null

sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli
containerd.io docker-buildx-plugin
docker-compose-plugin
# execute the MPE
sudo apt install -y unzip
wget https://zenodo.org/records/14854598/files/\
    wyunhao/Qelect-v3.2.zip
unzip Qelect-v3.2.zip
cd <directory_name_unzipped>
sudo docker build --no-cache -t ssle_project .
```

Communication setup To recover our estimated communication time under both LAN and WAN settings, we also need to equip all launched instances with proper files. Looking ahead, we simulate the communication time by letting them publish messages prepared beforehand. The following steps are for WAN setting. One could follow exactly the same procedure but only under AWS us-east-2 (Ohio) region for LAN setting.

1. In AWS console, under four AWS regions, execute the command in CloudShell (notice that this is a single-line command):

   ```
   aws ec2 describe-instances  --filters
   "Name=instance-state-name, Values=running"
   --instance-ids $instance_id  --query
   'Reservations[*].Instances[*].PublicIpAddress'
   --output text
   ```

2. Copy all 128 IPv4 public addresses returned from AWS CloudShell and put them in a single "ip.txt" file, one IPv4 address a line. Note that in LAN setting, we would have 128 IPv4 addresses directly for AWS us-east-2 (Ohio) region. Thus, when testing against WAN setting, pick 32 out of them.

3. Upload the "ip.txt" file from local to all instances via the following single-line command:

   ```
   pscp -p 100 -h ip.txt -x "-i  ssle.pem -o
   StrictHostKeyChecking=no" -l ubuntu ip.txt .
   ```

4. Dump random bytes to simulate the ciphertexts[1] used for broadcasting to all:

   ```
   dd if=/dev/urandom of=1M.txt bs=1M count=1
   ```

   And then also upload this simulated data to all instances:

   ```
   pscp -p 100 -h ip.txt -x "-i  ssle.pem -o
   StrictHostKeyChecking=no" -l ubuntu 1M.txt .
   ```

5. Upload the ssle.pem file from local to all instances via the following single-line command:

   ```
   pscp -p 100 -h ip.txt -x "-i  ssle.pem -o
   StrictHostKeyChecking=no" -l ubuntu ssle.pem .
   ```

[1] For $G = 128$, we have the communication size to be around 983KB. We use 1M data for simplicity.

6. Ssh to one of the instances and execute the following:

```
sudo apt-get update
sudo apt-get install -y build-essential
sudo apt -y install pssh
```

A.3.2 Basic Test

After installation, it should be easy to test the local computation, the evaluators could run the command:

```
After param generation.
Initial noise budget: 613
Lefted noise budget: 42
Total broadcast communication size ...: 67 MB.
--------------------------------
Total number of parties: 32
Preprocessed time      : 20446144 us.
Total time             : 2429615 us.
```

It shows that the local computation for one of the 32 parties takes around 2.4 seconds. The full log of our docker execution should include runtime for party sizes in $\{32, 64, 128, 256, 512, 1024, 2048\}$.

A.4 Evaluation workflow

The major results we want to reproduce here is the local computation and communication time for party size $G = 128$. To show evaluators the runtime trend, we consider a range of party sizes aforementioned.

A.4.1 Major Claims

(C1): The local runtime of $G \geq 2048$ is less than 10 seconds.

(C2): The communication time for $G = 128$ under LAN setting is around 25 seconds, and around 42 seconds under WAN setting. Since our communications happen in parallel, while the major prior work [1]'s communications run sequentially, we claim that our protocol is at least two orders of magnitude faster than theirs.

A.4.2 Experiments

(E1): *[Local Runtime] [<1 human-minute]*:

Preparation: Ssh to the instance with docker installed and execute the command

Execution: `sudo docker run ssle_project`

Results: The last log block should look like the following:

```
After param generation.
Initial noise budget: 613
Lefted noise budget: 47
Total broadcast communication size ... 4295 MB.
--------------------------------
Total number of parties: 2048
Preprocessed time      : 191495790 us.
Total time             : 6745102 us.
```

(E2): *[LAN Communication] [<1 human-minute]*:

Preparation: Follow the same steps as in Appendix A.3.1 when preparing for the communication tests, but use all 128 instances under the same region: AWS us-east-2 (Ohio). Ssh to one of the instances.

Execution:

```
MY_IP=$(curl https://ipinfo.io/ip)
time parallel-scp -p 100 -h ip.txt -x "-i \
 /home/ubuntu/ssle.pem -o StrictHostKeyChecking=no" \
 -l ubuntu 1M.txt "$MY_IP"_com.txt
```

Results: The final runtime log should be as follows. The final communication time is doubled since we have two rounds of broadcast. The evaluators could repeat the execution step multiple times to take the average.

```
real        0m2.192s
user        0m13.059s
sys         0m0.520s
```

(E3): *[WAN Communication] [<1 human-minute*

Preparation: Follow the same steps as in Appendix A.3.1 when preparing for the communication tests, with 32 instances under each of the four regions.

Execution:

```
MY_IP=$(curl https://ipinfo.io/ip)
time parallel-scp -p 100 -h ip.txt -x "-i \
 /home/ubuntu/ssle.pem -o StrictHostKeyChecking=no" \
 -l ubuntu 1M.txt "$MY_IP"_com.txt
```

Results: The final runtime log should be as follows. Same as before, we have two rounds in total and could take average of multiple trials for estimation.

```
real        0m8.772s
user        0m20.370s
sys         0m1.806s
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] D. Boneh, A. Partap, and L. Rotem. Post-Quantum Single Secret Leader Election (SSLE) From Publicly Rerandomizable Commitments. Cryptology ePrint Archive, Paper 2023/1241, 2023.

[2] Y. Wang and F. Zhang. Qelect: Lattice-based single secret leader election made practical. Cryptology ePrint Archive, Paper 2025/122, 2025.



USENIX Security '25 Artifact Appendix: H_2O_2RAM: A High-Performance Hierarchical Doubly Oblivious RAM

Leqian Zheng
City University of Hong Kong

Zheng Zhang
ByteDance Inc.

Wentao Dong
City University of Hong Kong

Yao Zhang
ByteDance Inc.

Ye Wu
ByteDance Inc.

Cong Wang
City University of Hong Kong

A Artifact Appendix

A.1 Abstract

This artifact contains the source code and scripts needed to reproduce our results. The Functional badge only requires an Ubuntu 20.04 environment with some easily installable packages. While reproducing the results presented in our paper, an Intel Xeon(R) Platinum 8457C processor and 2TB of RAM are necessary. If the specific CPU is unavailable, the results may exhibit minor deviations from the reported data. If memory is insufficient, a partial reproduction of our work for small datasets remains feasible. In addition, completing the full set of experiments requires several weeks, as the reported results are amortized over a substantial number of data accesses to more accurately assess the overhead of hierarchical ORAM.

A.2 Description & Requirements

Note that one can run H_2O_2RAM on a "non-confidential" machine to estimate the approximate performance overhead, as our design focuses on concealing memory access patterns rather than memory confidentiality and integrity (which are handled by the confidential virtual machine).

Software Requirements: Host operating system: Debian GNU/Linux 10 with the kernel version 5.15.120+. Confidential operating system: Ubuntu 20.04 with the same kernel version. Virtualization components: Qemu, OVMF, and Kata. Required tools: CMake (version $\geq$ 3.16), g++ (support C++-20), libboost, libtbb-dev, libnlopt-dev, libgtest-dev (for tests), Google Benchmark (for benchmarks), Python3.

Hardware Requirements: An Intel Xeon(R) Platinum 8457C processor (96-core) and 2TB of RAM. Confidential virtual machine needs 64 cores and 1TB of memory.

A.2.1 Security, privacy, and ethical concerns

This artifact will not lead to any security and privacy issues. One potential ethical concern is that running full experiment sets may take several weeks and result in (possibly notable) energy consumption.

A.2.2 How to access

The artifact is available on GitHub: https://github.com/55199789/H2O2RAM/commits/66d48b9 and Zenodo https://zenodo.org/records/14648338.

A.2.3 Hardware dependencies

We use and recommend a physical server powered by dual 48-core Intel Xeon(R) Platinum 8457C processors and 2TB RAM. Intel TDX, the specific TEE we use, is widely available with 5th Gen Intel Xeon Scalable processors. Alternatively, one can use other hardware-assisted CVMs, such as AMD SEV requiring AMD processors. Note that a "nontrusted" hardware (i.e., common CPUs) could also simulate our experiments as Intel TDX introduces only minor performance overheads for memory-intensive tasks (please refer to the work). Due to corporation policy, we do not support SSH access to the machine running our experiments.

A.2.4 Software dependencies

Setting up the TDX requires physical access to the machine (e.g., configuring the BIOS). We recommend following the instructions provided in the Intel TDX Module[1] and Intel TDX[2] for properly creating a TD image. In specific, we offer a guest image for CVM that includes the full experimental setup[3]. This image is derived from Ubuntu 20.04 and is preconfigured with TD guest, CMake, Google Benchmark, Boost, OpenMP, Intel TBB, and Google Test. Inside the image, use Python3 to run our experiment scripts.

A.2.5 Benchmarks

None.

[1] https://github.com/intel/tdx-module
[2] https://github.com/canonical/tdx
[3] https://zenodo.org/records/15000727/files/demo.zip

A.3 Set-up

A.3.1 Installation

Setting up the TDX is somewhat challenging. In case of any failure, you may skip related steps and continue directly on the Ubuntu 20.04 system.

Host Environment. Please follow the instructions in Intel TDX Module and Intel TDX to setup the environment. We also provide a sample scripts to enable Intel TDX in Host OS:

```
$ git clone https://github.com/canonical
    /tdx.git
$ cd tdx && sudo ./setup-tdx-host.sh
$ sudo reboot
```

Guest Image. Build `demo.raw` following this guide or download the pre-configured version. Then, run `./start.sh`. To boot in TD mode, uncomment lines 13∼15 of `start.sh`.

Software Dependencies. Log in to the TD VM and install all required packages:

```
$ git clone https://github.com/55199789/
    H2O2RAM.git
$ cd H2O2RAM && sudo ./setup.sh
```

To compile H_2O_2RAM:

```
$ mkdir build && cd build
$ cmake .. && make -j
```

A.3.2 Basic Test

Host Environment. Run `./25.tdx_host_check.sh` in the project directory. The output should include

```
[    22.593272] tdx: TDX module initialized.
```

TD VM. Log in to the TD VM and check that the memory encryption is active by `dmesg | grep -i tdx`. The output should look like:

```
[    0.000000] tdx: Guest detected
[    3.211618] Memory Encryption Features active: Intel TDX
[    13.230680] process: using TDX aware idle routine
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): H_2O_2RAM outperforms EnigMap by a factor ranging from 100× to 997× for oblivious map accesses with results reported in Figure 8 (a). H_2O_2RAM is compared with GraphOS for oblivious single-source shortest path computation as shown in Figure 8 (b), with GraphOS's results *taken* directly from its paper. Note that GraphOS and EnigMap neither compare nor cite each other. This claim is proven by the experiment E1.

(C2): H_2O_2RAM's performance under different scenarios (e.g., available threads, data block sizes). This is proven by the experiment (E2) discussed in §5.1 whose results are shown in Figure 7.

(C3): More precise failure probabilities of bucket hash tables & stashless Cuckoo hash tables. This is proven by the experiment (E3) discussed in §5.1 whose results are shown in Figure 5.

A.4.2 Experiments

(E1): *[Evaluation on different oblivious tasks] [10 human-minute + 6 compute-hour + 1 TB memory]:* A one-time preprocessing process will take about several compute-days. If your memory is insufficient, partial results will be produced.
How to: Under the project folder, run `cd benchmarks/ && python3 omap_exp_script.py` and `python3 osssp_exp_script.py` to obtain our results. In case of any interruption/corruption, please retry the Python scripts. Follow the instructions in the adapted repository to reproduce EnigMap's results. While GraphOS's results are taken directly from its paper, rather than being rerun in our environment.
Results: `benchmarks/result_omap_{n}.json` will include results for oblivious maps with n data blocks. `benchmarks/results_sssp.json` will include results for oblivious single-source-shortest-path computations across different graph sizes.

(E2): *[Evaluation on different scernarios] [30 human-minute + 12 compute-hour + 1 TB memory]:* A one-time preprocessing process will take about several compute-days. If memory is insufficient, partial results will be produced.
How to: Under the project folder, run `cd benchmarks/ && nohup python3 oram_exp_script.py &`. In case of any interruption/corruption, please try `nohup python3 oram_exp_script.py &` again.
Results: `benchmarks/result_{n}_{b}_{t}.json` includes results for n data blocks of size b with t threads.

(E3): *[Concrete failure probabilities] [10 human-minutes + 1 compute-hour]:* Leverage numeric methods to compute concrete failure probabilities of a) bucket hash tables for different bucket sizes ℓ and bucket numbers m, and b) stashless Cuckoo hash tables for different table sizes n and number of hash functions k.
How to: Under the project folder, run `cd failure_probabilities/ && ./compile.sh`. Then run the compiled binaries `./fail_prob_a` and `./fail_prob_b` respectively.
Results: `./fail_prob_a` will output ℓ versus different numbers of buckets. `./fail_prob_a` will output concrete failure probabilities δ with logarithmic values to the base 2 against different n and k.

A.5 Notes on Reusability

One can directly replace `std::vector` with H_2O_2RAM in their implementation to hide memory accesses. However, H_2O_2RAM

is not thread-safe even if all threads are performing only read operations, as reads can modify H_2O_2RAM's internal states.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.





USENIX Security '25 Artifact Appendix: OBLIVIATOR: OBLIVIous Parallel Joins and other OperATORs in Shared Memory Environments

Apostolos Mavrogiannakis*
UCSC

Xian Wang*
HKUST

Ioannis Demertzis
UCSC

Dimitrios Papadopoulos
HKUST

Minos Garofalakis
ATHENA Research Center &
Technical University of Crete

A Artifact Appendix

A.1 Abstract

We introduce oblivious parallel operators designed for both non-foreign key and foreign key equi-joins. Obliviousness ensures nothing is revealed about the data besides input/output sizes, even against a strong adversary that can observe memory access patterns. Our solution achieves this by combining trusted hardware with efficient oblivious primitives for compaction and sorting, and two oblivious algorithms: (i) an oblivious aggregation tree, which can be described as a variation of the parallel prefix sum, customized for trusted hardware, and (ii) a novel algorithm for obliviously expanding the elements of a relation. We then implemented our non-foreign key join and foreign key join with the two new algorithms. In the sequential setting, our oblivious join performs 4.6×-5.14× faster than the prior state-of-the-art solution (Krastnikov et al., VLDB 2020) on data sets of size $n = 2^{24}$. In the parallel setting, our algorithm achieves a speedup of up to roughly 16× over the sequential version, when running with 32 threads (becoming up to 80× compared to the sequential algorithm of Krastnikov et al.). Finally, our oblivious operators can be used independently to support other oblivious relational database queries, such as oblivious selection and oblivious group-by, which are implemented and evaluated with complex database queries.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There is no malicious or destructive operations in the artifact. We propose algorithms and implement systems with a positive impact on preserving data privacy. The artifact does not handle any sensitive data or personal information. All our tests were executed either on synthetic datasets or on publicly available real-world datasets. The ethical implications have also been carefully considered and our work is in full accordance with the 2025 USENIX Security ethics guidelines.

[1]The two first authors contributed equally to this work and their names are listed here alphabetically.

A.2.2 How to access

We open-source all code, datasets and scripts necessary for the evaluation of Obliviator and they can be accessed at [2].

A.2.3 Hardware dependencies

To evaluate the functionality of our artifact, we require a machine with an Intel processor and at least 9GB memory.

A.2.4 Software dependencies

Our implementations are for Linux OS. The artifact requires gcc-7 and g++-7. For all other software dependencies, e.g., the Open Enclave SDK [3], Intel SGX SDK [1], we provide scripts (./scripts/ae-install-dependencies.sh) to install them. In the script, the detailed package versions are chosen for the Ubuntu 20.04 environment.

A.2.5 Benchmarks

We provide ready-to-use datasets for some experiments in our paper. Other experiments need large datasets (hundreds of GB), and we provide scripts to download and process them. The datasets and scripts are in their corresponding sub-folders in ./data denoted by their dataset names. To evaluate the functionality of our artifact, the original hundreds of GB input datasets are not necessary and we provide a test input file (test.txt) within program folders (e.g., ./join/test.txt).

```
1  git clone [GitHub link to the artifact]
2  cd ./obliviator
3  chmod +x ./scripts/ae*
4  ./scripts/ae-install-dependencies.sh
5  source ~/.bashrc
6  ./scripts/ae-intel-sgx.sh
7  cd ~/linux-sgx
8  [Last commands printed in the terminal]
9  cd ~/obliviator
10 ./scripts/ae-basic-test.sh
```

Figure 1: Commands for set-up.

A.3 Set-up

A.3.1 Installation

To install the artifact and dependencies, we provide commands for reference in Figure 1. It may needs 1 hour in total (depending on detailed machines). After the execution of the command at line 6, another suggested command will be printed in the terminal. Please change the directory to another folder as suggested at line 7, and execute the command printed in the terminal. If there is any issue and error regarding the dependency installation, we suggest to execute the commands provided in ./scripts/ae-install-dependencies.sh and ./scripts/ae-intel-sgx.sh line by line to narrow down the detailed incorrect steps.

A.3.2 Basic Test

We provide ./scripts/ae-basic-test.sh to check compilation and see if all software dependencies are installed successfully. Please refer to line 9-10 in Figure 1. Error information will be printed in the terminal if it is not executed successfully.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We propose and implement the new oblivious parallel non-foreign key (NFK) join algorithm. Its function is to join two tables, i.e., matching table rows based on a given join attribute. Its performance with respect to consumed time is expected to be $2.86-77\times$ compared to KKS and KKS* [7], as reported in Section 5.1 of our paper (Table 2, Figure 9 and 10).

(C2): We propose and implement the new oblivious parallel foreign key (FK) join algorithm. Its function is to join a primary key table and a foreign key tables, where the join attribute in the primary table is distinct. Its performance with respect to consumed time is expected to be $1.7-60\times$ compared to Opaque [10], as reported in Section 5.1 of our paper (Table 2 and Figure 11 top left sub-figure).

(C3): We propose and implement the new oblivious filter operator. Its function is to retrieve specific subsets of elements from a database. Its performance with respect to consumed time is expected to be $2.8-52.3\times$ compared to Opaque, as reported in Section 5.2 of our paper (Figure 11 bottom sub-figure).

(C4): We propose and implement the new oblivious aggregation operator. Its function is to generate statistics or a summary of multiple table rows. Its performance with respect to consumed time is expected to be $2.5-53\times$ compared to Opaque, as reported in Section 5.2 of our paper (Figure 11 bottom sub-figure).

(C5): We also implement complex queries to test the above separate operators together. Their function is to obtain the final results of several given operators based on their detailed query requirements [4, 5]. Its performance with respect to consumed time is expected to be $2.58-37.57\times$ compared with Opaque, as reported in Section 5.2 of our paper (Figure 11 top right and bottom sub-figures).

```
1  ./scripts/ae-e1.sh
2  ./scripts/ae.sh
```

Figure 2: Commands for experiments.

A.4.2 Experiments

The evaluation consists of five experiments corresponding to the five claims in Section A.4.1. We provide a script (./scripts/ae.sh) for them, and please refer to the suggested commands in Figure 2 to execute the following five experiments. Alternatively, evaluators can run ./scripts/ae-e1.sh for E1 sub-experiment, ./scripts/ae-e2.sh for E2 sub-experiment, etc., separately.

(E1): [NFK Join] [5 compute-minutes + 9GB memory]: This experiment evaluates the NFK join of Obliviator, and compares it with KKS*. We are expected to see two floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator and KKS*. Our result is expected to be faster than KKS and KKS* and have the suggested speedup as introduced in C1 Section A.4.1. We also conduct E1 on a PC where Obliviator and KKS* take 0.0365 and 0.1294 seconds respectively.

(E2): [FK Join] [5 compute-minutes + 9GB memory]: This experiment evaluates the FK join of Obliviator, and compares it with Opaque. We are expected to see two floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator and Opaque. Our result is expected to be faster than Opaque and have the suggested speedup as introduced in C2 Section A.4.1. We also conduct E2 on a PC where Obliviator and Opaque take 0.0002 and 0.0004 seconds respectively.

(E3): [Filter Query] [5 compute-minutes + 9GB memory]: This experiment evaluates the filter quert of Oblivia-

tor, and compares it with Opaque. We are expected to see two floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator and Opaque. Our result is expected to be faster than Opaque and have the suggested speedup as introduced in C3 Section A.4.1. We also execute E3 on a PC where Obliviator and Opaque take 0.0076 and 0.1288 seconds respectively.

(E4): [Aggregation Query] [5 compute-minutes + 9GB memory]: This experiment evaluates the NFK join of Obliviator, and compares it with Opaque. We are expected to see two floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator and Opaque. Our result is expected to be faster than Opaque and have the suggested speedup as introduced in C4 Section A.4.1. We also execute E4 on a PC where Obliviator and Opaque take 0.1522 and 0.3788 seconds respectively.

(E5): [Complex Query] [5 compute-minutes + 9GB memory]: This experiment evaluates the NFK join of Obliviator, and compares it with KKS and KKS*. We are expected to see six floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator (the first three numbers) and Opaque (the last three numbers). Our result is expected to be faster than Opaque and have the suggested speedup as introduced in C5 Section A.4.1. We also executed E5 on a PC where Obliviator and Opaque spent 1.1705 (the sum of the first 3 numbers) and 3.4669 (the sum of the last 3 numbers) seconds respectively.

A.5 Notes on Reusability.

To reproduce the evaluation of the same experiments with hardware mode in our paper, large machines are required. We are unable to support the original evaluation here due to the large scale and our recent shortage of Intel SGX. The detailed original machine configuration is described in the Section 5 of our paper. Additionally, for reference, the detailed original configuration file for Intel SGX is given in ./script/parallel.conf. If users have an access to such large machines as our paper (i.e., Azure Standard_DC32ds_v3) and want to reproduce, please replace ./[program name]/enclave/parallel.conf with ./scripts/parallel.conf. And, for Obliviator, set the third parameter for oe_create_paralle_enclave in ./[program name]/host/parallel.c to be 0 (replacing the original OE_ENCLAVE_FLAG_SIMULATE), for KKS* (i.e., ./join_kks), set SGX_MODE in Makefile to be HW (or replace Makefile with the content in SGX_Makefile). Note that, to run KKS*, you need to further install the SGX PSW and Openssl, refer [1] or check ./scripts/intel-sgx2.sh .

We improve the scalability of the oblivious operators by designing efficient and fully parallel algorithms, and do not have extra optimizations for its paging overhead. We give our obliviousness proof in our extended paper [8]. And as mentioned in our paper, Intel Pin tool [6] and oblivious constant-time instructions from [9] are also used to help avoid software side-channel attacks for Intel SGX mentioned in our thread model Section 2.

When users test our programs with their own datasets, they are suggested to pay attention to the enclave configuration file parallel.conf and ensure the number of max threads, size of heap, stack (in parallel.conf) and MAX_BUF_SIZE (in parallel.c) are enough.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

References

[1] Intel(r) software guard extensions for linux* os. https://github.com/intel/linux-sgx.

[2] Obliviator. https://zenodo.org/records/15169905.

[3] Open enclave. https://github.com/openenclave/openenclave.

[4] Tpc-h benchmark. http://www.tpc.org/tpch.

[5] AMPLab, University of California, Berkley. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2014.

[6] Intel. Pin - a dynamic binary instrumentation tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[7] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *VLDB*, 2020.

[8] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. OBLIVIATOR: Oblivious parallel joins and other operators in shared memory environments. Cryptology ePrint Archive, Paper 2025/183, 2025.

[9] Nicholas Ngai, Ioannis Demertzis, Javad Ghareh Chamani, and Dimitrios Papadopoulos. Distributed & Scalable Oblivious Sorting and Shuffling . In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4277–4295, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[10] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, 3 2017. USENIX Association.



USENIX Security '25 Artifact Appendix: Efficient Ranking, Order Statistics, and Sorting under CKKS

Federico Mazzone
University of Twente

Maarten Everts
University of Twente
Linksight

Florian Hahn
University of Twente

Andreas Peter
Carl von Ossietzky
Universität Oldenburg

A Artifact Appendix

A.1 Abstract

The artifact consists of the complete codebase supporting our paper, which introduces a novel approach to performing ranking, order statistics, and sorting under CKKS, a fully homomorphic encryption scheme. Specifically, the artifact includes a C++ implementation of the four functionalities discussed in the paper: ranking, argmin, median, and sorting, all built on top of the OpenFHE library for CKKS.

This implementation was designed to benchmark the runtime of these functionalities, the primary empirical result of our paper, across vectors of increasing length in both single- and multi-threaded settings.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We do not identify any security risks associated with installing or using our artifact.

A.2.2 How to access

We have made our artifact available on multiple platforms:

- As permanent storage on Zenodo: https://doi.org/10.5281/zenodo.14673904
- As working repository on Github: https://github.com/FedericoMazzone/openfhe-statistics
- As docker container on Docker Hub: https://hub.docker.com/r/mazzonef/openfhe-statistics

A.2.3 Hardware dependencies

No specific hardware is required, as our library runs exclusively on CPU. OpenFHE and CKKS require significant RAM, particularly for larger experiments (see Table 2 in our paper). In the largest experiment (sorting with 16,384 elements), memory consumption may reach up to 230 GB.

To replicate our performance results, we used a machine equipped with an Intel Xeon Platinum 8358 processor running at 2.60 GHz, featuring 32 cores (64 threads) and 512 GB of RAM.

A.2.4 Software dependencies

Our implementation relies on the OpenFHE library, available at https://github.com/openfheorg/openfhe-development. We specifically use version 1.1.2, which is included in our repository to avoid compatibility issues.

Building and running our implementation requires a C++ compiler and CMake. We recommend using GCC v10 or Clang 11 on Linux for best compatibility.

We have tested our repository exclusively on Linux Ubuntu. While it should theoretically compile and function on Microsoft Windows, we do not guarantee compatibility.

A.2.5 Benchmarks

We use synthetic data points generated in the range [0,1] with a precision of two decimal places. The exact dataset is available in the file `data/points1d.csv`.

A.2.6 Benchmarks

We use synthetic data points generated in the range [0,1] with a precision of two decimal places. The exact dataset is available in the file `data/points1d.csv`.

A.3 Set-up

A.3.1 Installation

Unless using the Docker container, both the OpenFHE library and our library must be compiled. Installation steps are detailed in the README file, but a summary is provided below:

1. Install prerequisites:
   (a) `sudo apt-get install build-essential`
   (b) `sudo apt-get install cmake`
2. Install OpenFHE:

(a) `cd openfhe-development-1.1.2`

(b) `mkdir build`

(c) `cd build`

(d) `cmake ..`

(e) `make -j`

(f) `sudo make install`

(g) `cd ../..`

3. Install our library:

(a) `mkdir build`

(b) `cd build`

(c) `cmake ..`

(d) `make -j`

(e) `cd ..`

If you do not have sudo access in your machine, specify a different installation path for OpenFHE. See more about this in the README file.

A.3.2 Basic Test

To run a basic test, execute `./build/demo`. This runs a demonstration of all four functionalities on a fixed vector. The expected console output includes initialization of the CKKS cryptosystem, followed by the execution of ranking, argmin, median, and sorting in sequence.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our four functionalities (ranking, argmin, median, and sorting) exhibit asymptotic quadratic runtime, achieving the runtimes reported in Figure 6 and memory consumption reported in Table 2 of the paper. This is validated by the experiment (E1) described in Section 6 of the paper.

A.4.2 Experiments

(E1): Main experiment, 1 human-minute + 717 compute-hour + 230GB memory
Preparation: Compile and install the library.
Execution: Run the command `sh ./benchmark.sh <algorithm>`, where `algorithm` is one of the following: ranking, ranking-tie, minimum, median, sorting. The script will automatically execute tests on vectors of increasing length (from 8 to 16,384) in both single- and multi-threaded settings.
Results: Runtime and memory consumption results are stored in the CSV file `benchmark.out` (one row per experiment). On a comparable machine, performance metrics (runtime and memory) should be similar to those reported in the paper. Note that running the Docker version may introduce runtime overhead, particularly for short vectors.

A.5 Notes on Reusability

The parameters of our tests are set to work with data with two decimal digits precision. In case the input data needs to be more or less precise, the comparison approximation parameters need to be adjusted accordingly. See more about this in the README file.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.