University of California
Santa Barbara

A System-level Framework for Privacy

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Deeksha Dangwal

Committee in charge:

Professor Timothy Sherwood, Chair
Professor Chandra Krintz
Professor Jonathan Balkind
Professor Sandhya Dwarkadas

December 2023

The Dissertation of Deeksha Dangwal is approved.

Professor Chandra Krintz

Professor Jonathan Balkind

Professor Sandhya Dwarkadas

Professor Timothy Sherwood, Committee Chair

March 2022

A System-level Framework for Privacy

To Mom, Dad, Anu, and Nani

In Memory of Raja Ram Dangwal, Bimla Devi Dangwal,

Naveen Dangwal, Abha Dangwal, and Ram Mohan Misra

“Methought that of these visionary flowers

I made a nosegay, bound in such a way

That the same hues, which in their natural bowers

Were mingled or opposed, the like array

Kept these imprisoned children of the Hours

Within my hand,—and then, elate and gay,

I hastened to the spot whence I had come,

That I might there present it!—Oh! to whom?”

Percy Bysshe Shelley

Acknowledgements

Mom, and Dad, you both have been unwavering pillars of support throughout this academic journey (truly this PhD belongs to you as much as it does to me). Your perennial belief in me, especially on days I doubted myself the most, has meant everything to me. You have grounded me on the most trying days, and I can't thank you enough for reminding me to enjoy the ride. I couldn't have done this without you. Animesh, thank you for your curiosity and genuine interest in my research. Your enthusiastic questions kept my excitement alive even when I felt disheartened and uninspired. Thank you also for serving as my rubber duck: I often wrote my paper introductions as if I were explaining my research to you, so thank you for being such a good listener. ;)

Thank you to the best advisor anyone could ever ask for: Timothy "Processor-of-Computer-Science" Sherwood. Tim, there will never be enough words to express how much I value your mentorship (read: I am *this* close to turning these acknowledgements into a Tim tribute). Tim is WACI (has Wild And Crazy Ideas) and his top-notch research on the "fringe" has taught me to connect ideas from "far away" fields and has brought this interdisciplinary thesis to life. Tim is miraculously both hands-on and hands-off as an advisor and gives his students the best possible educational experience. I will hold steadfastly to the *Tim-isms* I have learned. I still remember sitting in Tim's CS254 lecture on RISC vs. CISC processors and wondering if he would ever take me on as his student. I am so thankful you did, Tim. You are the reason I feel like I belong in computer science. I will miss hearing about the latest python packages and podcast finds (e.g. on how chimpanzees communicate), references to thinking fast and slow (and other great books), and of course, the *Rockwell Retro Encabulator*. You are a great advisor because you are a great person. I have not only learned how to be an exceptional scientist, but also how to be an exceptional human being from you. Thank you for your kindness and unequivocal support throughout and thank you for the opportunities you have brought my way.

Thank you to my committee members. Chandra, your CS263 remains one of my favorite classes because I went in an electrical engineer and emerged a computer scientist. I loved the lectures, our after-class discussions, and the projects. Thank you also for guiding so much of my thesis. Jon, thank you for being my guide into the computer architecture community. You have been so kind and supportive; so much that I don't think either of us will forget my ASPLOS talk (read: Jon heard it $\tilde{3}0$ times). Sandhya, you have been such a great role model and mentor, thank you for making me feel so welcome in our community. Ever since we met in CRA-W Grad Cohort in 2017, I have walked away feeling more confident in myself and our community. Another important thanks goes to Prof. Michael Melliar-Smith, who encouraged me to change my major and was later the first person to suggest I apply to the Ph.D. program. Your lectures and stories about Cray supercomputers and stories about the Acorn RISC Machine were so interesting that I ended up taking another graduate class in computer architecture which changed the trajectory of my career. Thank you.

Thank you to *my village.* Starting with the best lab in the entire world with the best people ever: the ArchLab, my home in UCSB. Joseph and Weilong, you showed me what it's like to *be* a graduate student. From our late nights in the lab during deadline season eating Poke and Naan Stop, to lab taco Tuesdays at Brophy's, and all the following shenanigans in between: game nights (including spooky Halloween candle games), getting our machines compromised by international hackers, talking about uncertainty, and monads, and working on SLMs (small language models) for Arch-a-thons, I had the best time with you. Thank you for teaching me your ways and setting such great examples. Alvin, thank you for being my pandemic buddy. Apart from working on our projects, chatting with you about Obsidian, traveling in Asia, and all the keyboards and monkeytyping was a much needed respite during a very tumultuous time. Vaibhav, Sujaya, Victor, and Nevena, thank you for your camaraderie and for all the happy memories we made during happy hours.

Thank you to the UCSB community; it is rare to feel like you belong to an institution that truly cares about you, but this is what I felt like at UC Santa Barbara. I owe great thanks to the Department of Electrical and Computer Engineering and the Department of Computer Science for all the opportunities, guidance, excellent education and fantastic research support, and for fostering an inclusive and fun environment for your students. Thank you especially to Val, Benji, Jill, Karen, and Samantha for your support throughout. I had the most incredible opportunity to mentor many students at UCSB throughout my PhD career and I learned so much from them. Thank you Saurabh Gupta, Jacqueline Mai, Dawit Aboye, Dylan Kupsh, Maggie Lim, Junayed Naushad, Manu Kondapaneni, and Bisman Sodhi. Working with you all was the most rewarding part of my graduate experience. Thank you Diba for making so much of this rewarding process possible.

I have many people to thank from three great internship experiences and I'll do this in one breath. From Reality Labs Research, I'd like to thank Vincent Lee, Armin Alaghi, Hyo Jin Kim, Tianwei Shen, Meghan Cowan, Rajvi Shah, Caroline Trippel, Brandon Reagen, Vasileios Balntas, Eddy Ilg, Richard Newcombe, and Sarah Rathbun; from Microsoft Research, I'd like to thank Doug Burger, Eric Chung, Karin Strauss, and Andrew Putnam; and from Oracle Labs, thank you Ryan Bedwell and Sasitharan Murugesan. You all took a chance on me and helped me work on great projects.

Thank you to my lifelines: Itir and Camille. I cannot imagine life without the dreamy days of San Remo. You have been my most staunch supporters and going through the ups and downs of graduate school life together has been a privilege; I will forever cherish our homely antics. Anna, thank you for modeling strength and independence, and teaching me the power of genuine curiosity. KP, Argya, Preethi, thank you for being my constants, my connection to my roots, and my reminders of how far I have come.

Thank you most of all to Abhe. Together we have seen the best of times (and we have seen the worst of times). You challenged me to expand my thinking (learn signal processing) and showed me how to dream big. Our early conversations on the beach reminded me how much I love science and I know today that there's no one else I'd rather spend time solving problems with. Thank you for always being there for me; both when I wanted to brainstorm and when I wanted to vent. Thank you for sharing so much with me: your wild ideas and tips and tricks and python functions and servers you built and so much more. Thank you for being my partner in this amazing journey. Thank you for always picking me up when I was down and for celebrating with me when life was good. Thank you for showing me *Pollo Fino.* You get me, and I cannot wait for a lifetime of dreaming big with you.

To all the people who stood up for me when I was not around, to all the people who took a chance on me, to all my coauthors, colleagues, friends, and family: THANK YOU. ***"Maybe the Best Paper Award was the friends we made along the way"***.

Curriculum Vitæ
Deeksha Dangwal

Education

2022	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2016	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2014	B.E. in Instrumentation Engineering, Ramaiah Institute of Technology, Bangalore, India

Experience

2015-2022	*Graduate Student Researcher*, UCSB ArchLab
2020-2021	*Research Intern*, Facebook Reality Labs Research, Redmond, WA
2018	*Research Intern*, Microsoft Research, Redmond, WA
2016	*Research Assistant*, Oracle Labs, Austin, TX

Honors and Awards

2021	*Grad Slam Runner-Up*, UC Santa Barbara
2020	*Rising Star in EECS*, UC Berkeley
2020	*IEEE Micro Top Pick* "Trace Wringing for Program Trace Privacy"
2020	*Fiona and Michael Goodchild Graduate Mentoring Award*, Graduate Division, UC Santa Barbara
2020	*Outstanding Graduate Student Award*, Department of Computer Science, UC Santa Barbara

Conference and Journal Publications

PLDI 2021	"Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption", Meghan Cowan, **Deeksha Dangwal**, Armin Alaghi, Caroline Trippel, Vincent T Lee, Brandon Reagen in Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation [1]
BMVC 2021	"Mitigating Reverse Engineering Attacks on Local Feature Descriptors", **Deeksha Dangwal**, Vincent T Lee, Hyo Jin Kim, Tianwei Shen, Meghan Cowan, Rajvi Shah, Caroline Trippel, Brandon

	Reagen, Timothy Sherwood, Vasileios Balntas, Armin Alaghi, Eddy Ilg in the 32nd British Machine Vision Conference (BMVC) [2]
SEED 2021	“Context-Aware Privacy-Optimizing Address Tracing”, **Deeksha Dangwal**, Zhizhou Zhang, Jedidiah R Crandall, Timothy Sherwood in IEEE International Symposium on Secure and Private Execution Environment Design (SEED) [3]
IEEE Micro 2020	“Trace Wringing for Program Trace Privacy”, **Deeksha Dangwal**, Weilong Cui, Joseph McMahan, Timothy Sherwood in IEEE Micro Top Picks 2020 [4]
IEEE Micro 2020	“Agile Hardware Development and Instrumentation With PyRTL”, **Deeksha Dangwal**, Georgios Tzimpragos, Timothy Sherwood in IEEE Micro [5]
JETC 2019	“Language Support for Navigating Architecture Design in Closed Form”, Weilong Cui, Georgios Tzimpragos, Yu Tao, Joseph McMahan, **Deeksha Dangwal**, Nestan Tsiskaridze, George Michelogiannakis, Dilip P Vasudevan, Timothy Sherwood in ACM Journal on Emerging Technologies in Computing Systems (JETC) [6]
ASPLOS 2019	“Safer Program Behavior Sharing through Trace Wringing”, **Deeksha Dangwal**, Weilong Cui, Joseph McMahan, Timothy Sherwood in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems [7]
ISCA 2018	“Charm: A Language for Closed-Form High-Level Architecture Modeling”, Weilong Cui, Yongshan Ding, **Deeksha Dangwal**, Adam Holmes, Joseph McMahan, Ali Javadi-Abhari, Georgios Tzimpragos, Frederic Chong, Timothy Sherwood in 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA) [8]
FPL 2017	“A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation”, John Clow, Georgios Tzimpragos, **Deeksha Dangwal**, Sammy Guo, Joseph McMahan, Timothy Sherwood in 27th International Conference on Field Programmable Logic and Applications (FPL) [9]

Workshop Publications

HASP 2020	“SoK: Opportunities for Software-Hardware-Security Codesign for Next Generation Secure Computing”, **Deeksha Dangwal**, Meghan Cowan, Armin Alaghi, Vincent T Lee, Brandon Reagen, Caronline Trippel in Hardware and Architectural Support for Security and Privacy (HASP) [10]
WCAE 2019	“PyRTL in Early Undergraduate Research”, Diba Mirza, **Deeksha Dangwal**, Timothy Sherwood in Proceedings of the Workshop on Computer Architecture Education [11]
EMC2 2019	“PyRTLMatrix: An object-oriented hardware design pattern for prototyping ML accelerators”, Dawit Aboye, Dylan Kupsh, Maggie Lim, Jacqueline Mai, **Deeksha Dangwal**, Diba Mirza, Timothy Sherwood in the 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2) [12]

US Patents

2020	“Deriving a concordant software neural network layer from a quantized firmware neural network layer”, Jeremy Fowers, Daniel Lo, **Deeksha Dangwal** [13]

Abstract

A System-level Framework for Privacy

by

Deeksha Dangwal

Privacy in the digital age has become increasingly difficult to achieve. While there is consensus on the importance of building privacy into systems that deal with sensitive information, our ability to reason about system-level privacy is severely limited. In this work, I introduce wringing, a new computer architecture approach for building privacy in systems to minimize information leakage. I detail how wringing enhances the privacy of program traces and how it opens up a new optimization space between privacy and utility.

Next, I demonstrate how wringing generalizes beyond traces: in computer vision pipelines that rely on streaming user data for localization tasks in augmented reality settings. We discover a new reverse engineering attack on localization pipelines that can compromise user privacy and show that data minimizing wringing serves as a mitigation for such attacks.

Finally, I present a new architecture that builds privacy into personal devices. Our architecture supports both data minimizing techniques like wringing and differential privacy to protect streaming data being crowd-sourced by a central aggregator. With this hardware implementation, we can enforce the user's privacy settings and prevent unintended data leakage.

Contents

Curriculum Vitae **x**

Abstract **xiii**

1 Introduction **1**
- 1.1 Thesis Statement 5
- 1.2 Research Summary and Overarching Theme 6
- 1.3 Permissions and Attributions 9

2 Trace Wringing for Safer Program Behavior Sharing **11**
- 2.1 Introduction 12
- 2.2 Wringing a trace 15
- 2.3 Related work 18
- 2.4 Our approach to wringing 22
- 2.5 Evaluation 31
- 2.6 Conclusion 41

3 Context Matters: Optimizing Privacy of Traces with Information Flow Tracking **44**
- 3.1 Introduction 45
- 3.2 Modeling Threats to Privacy of Traces 49
- 3.3 Scrubbing Data from Traces 54
- 3.4 Evaluation 62
- 3.5 Related Work 73

4 Wringing Beyond Traces: Mitigating Reverse Engineering Attacks in Computer Vision Pipelines **78**
- 4.1 Introduction 79
- 4.2 System and Threat Definition 82
- 4.3 Background 85
- 4.4 Reverse Engineering Attack 87
- 4.5 Evaluation 92

4.6 Related Work 101
4.7 Conclusion 104

5 A Privacy-Enhancing Architecture for Crowd Sourced Data 105
5.1 Introduction 106
5.2 Privacy Threat Model 110
5.3 Privacy-Enhancing Architecture 112
5.4 Private Stream Generation 117
5.5 Hardware Architecture 127
5.6 Evaluation 129
5.7 Related Work 134
5.8 Conclusion 137

6 Conclusion and Future Directions 139
6.1 Future Directions 140

Bibliography 143

Chapter 1

Introduction

Privacy in the digital age has become increasingly difficult to achieve as technologies that capitalize on sensitive information such as facial recognition, location services, health tracking, etc. have become mainstream. Policymakers have put in place regulations on data protection including the European General Data Protection Regulation (GDPR), California Consumer Privacy Act (CCPA), Illinois Biometric Information Privacy Act (BIPA), etc. and while there is growing consensus on the importance of building privacy into systems that deal with sensitive information, our ability to reason about and implement system-level privacy is severely limited so far. As a field, we must develop tools, mechanisms, and primitives to uphold these regulatory protections.

In this thesis, I introduce a new framework for privacy that is especially productive when considering the whole computer system. This includes considering various system-level and application-specific concerns, including performance, power usage and availability, privacy algorithm design, and the definition of privacy itself. For the applications presented in this thesis, the methods simultaneously consider software, hardware, and privacy design parameters when optimizing for power- and performance-efficient, high-fidelity, and threat-model-optimal solutions. Like many computer systems problems, there exist several tradeoffs that we must manage, and in this work, I

It is natural to extend this idea of having a tightly coupled, synergistic codesign feedback loop to include security constraints. We define the notion of software-hardware-security codesign as Our goal is to establish a set of design consideration dependencies for each secure computing technology, and expose the feedback loops between the nodes and the opportunities for enabling iterative codesign.

Having studied this tradeoffduring my phd, I have found that there is a recipe for success to design private systems efficiently.

The ingredients are: a means to minimize the information leakage A means to traverse and optimize the privacy-utility trade off space A means to enforce privacy-enhancing methods This thesis is interdisciplinary and contains ideas from computer architecture, computer vision, and security and privacy.

I introduce wringing, a new computer architecture approach to building privacy in systems to minimize information leakage while still maintaining utility of the privatized data. When working towards application-tuned systems, developers often find themselves caught between the need to share information (so that partners can make intelligent design choices) and the need to hide information (to protect proprietary methods or sensitive data). One place where this problem comes to a head is in the release of program traces, for example a memory address trace. A trace taken from a production server might expose details about who the users are or what they are doing, or it might even expose details of the actual computation itself (e.g. through a side channel). To protect the privacy of user data and safeguard proprietary details, engineers are often asked to make, by hand, "analogs" of their codes that would be free from such sensitive data or, may even try to describe behaviors at a high level with words. These approaches lead to missed opportunities, confusion, and frustration. We propose a new problem for study, trace wringing, that seeks to remove as much information from the trace as possible while still maintaining key characteristics of the original. We formalize this problem and show that, for a specific instance around memory traces, as little as a few thousand bits need to be shared. We demonstrate experimentally that the trace-wrung proxies behave similarly in the context of cache simulation but with bounded leakage, and examine the sensitivity of wrung traces to a class of attacks on AES encryption.

As global policy progressively moves in favor of protecting user data, and the ability to gather and process very personal information at scale becomes commonplace, it becomes all the more pressing to find and develop methods that prioritize user and data privacy. Computer architecture research is already enmeshed in security research as a result of recently exposed architecture vulnerabilities. But we must also address the rising importance of privacy as a field. Fortunately, we do not have to start from scratch and can build on top of methods such as LINDDUN [14] privacy threat modeling, which we apply to the problem of privately sharing traces. While prior approaches look primarily at bounding leakage through the uniform application of an extreme form of lossy compression, trace scrubbing represents a new family of approaches that attempt to remove information in a more targeted fashion. This can range from a simple redaction of certain addresses (when the sensitive information has very limited footprint in the trace) to techniques that attempt to intelligently apply a mixture of methods to minimize the number of bits leaked while attempting to achieve the best possible utility for memory access traces. We present the cache and prefetch performance of our strategies and, by only scrubbing portions of a trace where sensitive information might flow, we place an upper bound on information leakage that surpasses prior work in this area by an order of magnitude. Trace scrubbing accurately captures trace behavior while minimizing leakage. By connecting this problem to information flow tracking techniques in the context of scrubbing execution traces, we open the door to even further refinement through the application of more advanced information flow analysis techniques and the adoption of multi-level privacy schemes that can further limit the amount of potential leakage.

In the next chapter, we demonstrate how wringing generalizes beyond address traces. The widespread use and the low cost of sensor systems, photography, and video capture has unlocked new computational and algorithmic approaches to health, entertainment, transportation, robotics, and many other fields. As applications such as autonomous driving and augmented reality evolve, a practical concern is data privacy. Specifically, we look at applications that rely on localization based on user images. The widely adopted technology uses local feature descriptors, which are derived from the images and it was long thought that they could not be reverted back. However, recent work has demonstrated that under certain conditions reverse engineering attacks are possible and allow an adversary to reconstruct RGB images. This poses a tremendous risk to user privacy. We take this a step further and model potential adversaries using a privacy threat model. Subsequently, we show under controlled conditions a reverse engineering attack on sparse feature maps and analyze the vulnerability of popular descriptors including FREAK, SIFT and SOSNet. Finally, we evaluate potential mitigation techniques based on wringing and scrubbing that select a subset of descriptors to carefully balance privacy reconstruction risk while preserving image matching accuracy; our results show that similar accuracy can be obtained when revealing less information.

Finally, we present an architecture that enables data minimizing wringing for privacy of streaming data in a crowd-sourcing application. For stronger guarantees on the privacy of users' biometric information, our architecture supports the addition of differentially private noise. Biometric information collected from wearable devices can yield new insights with the potential to improve the health and wellness of those individuals under measurement. The aggregation of this data over ever larger groups compounds this potential benefit by helping professionals understand the full shape of the distribution, however the nature of such biometric data is extremely personal. Prior work has shown how such shared data can leak your gender, age, habits, and can even be linked back to identity. Future computer architectures have a role to play in protecting user privacy, and we find their use in addressing the privacy loss associated with sharing time-series data specifically (such as those collected from wearables) to be most critical. We introduce the two-stream privacy architecture including two privacy-enhancing interventions that, through a small-footprint hardware extension, can both bound the amount of information leaving a user's wearable device and provide differential privacy guarantees. Through a careful formulation of privacy as an architectural design constraint, the examination of interacting privacy-enhancing parameters, a hardware design and evaluation, and the evaluation of privacy versus utility for a suite of privacy-sensitive applications, we show a flexible and effective privacy framework enabling sharing of streaming sensor data.

1.1 Thesis Statement

There is consensus on the importance and urgency to build privacy into computer systems that deal with sensitive information. However, system designers and architects have historically focused on improving system performance and do not have reliable tools and methods to balance privacy and utility.

In this thesis, we demonstrate and explore a fundamental tradeoff between privacy and utility in many computer systems problems. Within the context of these problems, we unify privacy-enhancing interventions such as data minimization, anonymization, and differential privacy. We further introduce application-dependent system-level metrics to evaluate privacy and utility and leverage these to achieve optimal privacy and utility design points. Finally, we enforce these interventions with computer architecture innovations.

1.2 Research Summary and Overarching Theme

The main thrust of this thesis is a new approach to privacy: wringing. The essence of system design is in maneuvering tradeoffs and it continues to be the bottom line in this work on the design of private systems as well. Once *privacy* and usefulness of privatized data, *utility*, is defined, wringing is used to reason about privacy-preserving choices and traversing the *privacy-utility tradeoff* space. In this thesis, we see this problem come to a head in the release of program traces for hardware-software co-optimization, in visual pipelines which form the backbone of always-on visual systems such as augmented reality and autonomous driving, and finally in aggregating crowd-sourced biometric data.

My research on trace wringing [7, 4] navigates this quandary by sharing the *structure* of the program trace without leaking the *actual addresses*. Trace wringing is a new paradigm of anonymity and privacy of traces where compression and modeling provide a way to release information with easily verifiable bounds on leakage. Trace wringing uses a surprisingly simple metric to quantify information leakage—*number of bits*, and estimates utility through cache simulations. Trace-wrung proxies leak as few as tens of thousands of bits which is *orders of magnitude fewer* than prior work on compressed traces and even profiles used in synthetic trace generation. To evaluate the security of trace wringing beyond bit leakage, I show that a class of existing AES attacks fails to find useful information in the leakage-bounded proxies. Trace wringing introduces a tradeoff between privacy and utility and each proxy trace is a point in this bit-error space. Various points in the bit-error tradeoff space can be compared with each other and the "best" points can be used for sharing program behaviors safely and effectively. However, there is a complex space of possibilities to consider and a systematic framework is required. I have subsequently shown that it is possible to formulate trace privacy as a non-gradient optimization problem to better explore the bit-error tradeoff space and this has driven improvements of an order of magnitude in both leakage and accuracy.

Trace scrubbing further links trace privacy to the well-studied problem of information flow analysis. When naively wringing a trace, there is no sense of *where* the most sensitive data is and all addresses in the trace are considered equally “bad to leak”. Accordingly, the leakage of all the addresses must be restricted to a similar degree to bound the overall leakage. However, in real applications, some trace data may be under the influence of extremely sensitive data while others may not. In fact, in many cases, only a small subset of the entire set of addresses accessed by an application may be related to private information. This is the key insight: *not all addresses in a program trace are equally bad to leak*. Now, sensitive information can be identified at the program level, and its impact can be determined in the resulting trace. When only a subset of the addresses in a trace are deemed sensitive, then one can either eliminate or reduce leakage from those addresses specifically. I present an ensemble of mitigation or *scrubbing* techniques that at the extreme end simply delete or *redact* sensitive addresses from a trace or replaces sensitive data with leakage-reduced addresses that are behaviorally similar.

For systems that have direct access to user data through their personal devices, user privacy is a practical and urgent concern. One such setting is commonly seen in always-on augmented reality devices that rely on user images and video to perform localization and mapping tasks through feature descriptors. In this thesis, I re-examine the privacy of systems which shared user image local descriptors with the server [15]. **I first showed that these descriptors, previously considered private, can be used to reverse engineer raw user images.** I designed and trained a generative adversarial network (GAN) that attacked the local descriptors. **My results surpassed the state-of-the-art reconstruction accuracy**. Next, I provided mitigation techniques to protect the user's data. Yet again, I employed the **privacy-utility tradeoff to reason about privacy-enhancing design choices and was able to show that optimal solutions**

which maximize privacy and accuracy do, in fact, exist. My mitigation techniques minimized the leakage of data, by sharing fewer features. This is either done by treating all features as equally leaky (similar to trace wringing), or specifically targeting features around sensitive categories such as people, license plates, etc. (similar to trace scrubbing). Another significant contribution of this work was proposing a privacy threat modelling procedure in the computer vision and machine learning domain. **I also introduced a new privacy metric: *semantic privacy***, based on the similarity of objects detected on both the original image and the reconstructed privacy-maximized images using YOLOv3 [16].

Privacy is an increasingly critical consideration of system design, and while multiple large corporations have started to invest in privacy preserving technologies, there is still a great deal of room for innovation. Techniques that distribute the responsibility of privacy and avoid centralized points of naked aggregation are useful both because they lower the responsibility of aggregators and because they avoid single points of failure. Even if we are to trust a few entities with our most private data, there is now (and likely always will be) an appetite for our data beyond our ability to carefully examine. We propose the Two-Stream Privacy architecture, a new framework for supporting privacy management that combines information theoretic methods with randomized response-based local different privacy to enable private aggregation of wearable time-stamped sensor data. Key to this solution is a privacy-preserving Data Minimization Unit which uses Short-Time Fourier Transform that allows mutual information reduction by using a filter for lossy reconstruction of input signals. As a demonstration, we successfully evaluate the effectiveness of our technique for sensor data such step count and BPM from real world accelerometer and electrocardiagram sensor readings. Over all we find that a carefully reduced disclosure, when coupled with random response, can unlock parts of the design space not reachable by random response alone. In several cases the error could be reduced by a factor of 3x or more under the same privacy budget and have up to 72% improvement in privacy for the same utility tolerance. Furthermore we find that the hardware overhead of such an implementation is quite small and we find the proposed solution does not have significant overhead in terms of chip area and power consumption.

1.3 Permissions and Attributions

1. The content of Chapter 2 is the result of a collaboration with Weilong Cui, Joseph McMahan, and Timothy Sherwood. Parts of this chapter have previously appeared in the Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems [7] and IEEE Micro "The 2019 Top Picks in Computer Architecture", Volume 40, Issue 3 [4]. It is reproduced here with the permission of ACM [1] and IEEE [2].

2. The content of Chapter 3 is the result of a collaboration with Zhizhou Zhang, Jedidiah Crandall, and Timothy Sherwood. It has previously appeared in the 2021 International Symposium on Secure and Private Execution Environment Design (SEED). It is reproduced here with the permission of IEEE [3].

3. The content of Chapter 4 is the result of a collaboration with Vincent T. Lee, Hyo Jin Kim, Tianwei Shen, Meghan Cowan, Rajvi Shah, Caroline Trippel, Brandon Reagen, Timothy Sherwood, Vasileios Balntas, Armin Alaghi, Eddy Ilg. Parts of this work have appeared in the 2021 British Machine Vision Conference [2].

4. The content of Chapter 5 is the result of a collaboration with Alvin Glova, Abhejit Rajagopal, Rhys Gretsch, Pranjali Jain, Jonathan Balkind, and Timothy Sherwood.

[1]https://authors.acm.org/main.html
[2]https://www.ieee.org/content/dam/ieee-org/ieee/web/org/pubs/permissions_faq.pdf

5. The content of Chapters 1 and 6 is a result of all the collaborations mentioned above. Parts of these chapters have previously appeared in the conferences, journals, magazines mentioned above and in the 2020 workshop on Hardware and Architectural Support for Security and Privacy (HASP) [10]. It is reproduced here with the permission of ACM and IEEE.

Chapter 2

Trace Wringing for Safer Program Behavior Sharing

In this chapter, I introduce the computer architecture method of wringing and apply wringing to traces to minimize information leakage when sharing program traces for application tuning. When working towards application-tuned systems, developers often find themselves caught between the need to share information (so that partners can make intelligent design choices) and the need to hide information (to protect proprietary methods or sensitive data). One place where this problem comes to a head is in the release of program traces, for example a memory address trace. A trace taken from a production server might expose details about who the users are or what they are doing, or it might even expose details of the actual computation itself (e.g. through a side channel). Engineers are often asked to make, by hand, "analogs" of their codes that would be free from such sensitive data or, may even try to describe behaviors at a high level with words. Both of these approaches lead to missed opportunities, confusion, and frustration. We propose a new problem for study, trace wringing, that seeks to remove as much information from the trace as possible while still maintaining key characteristics of the original. Trace wringing exposes a new tradeoff space between privacy and utility in the context of address traces. We formalize this problem and show that, for a specific instance around memory traces, as little as a few thousand bits need to be shared. We demonstrate experimentally that the trace-wrung proxies behave similarly in the context of cache simulation but with bounded leakage, and examine the sensitivity of wrung traces to a class of attacks on AES encryption.

2.1 Introduction

A quantitative approach to optimizing computer systems requires a good understanding of the way applications exercise a machine; real program traces taken from production code, in production environments lead to the clearest understanding. Unfortunately, even the simplest program traces, such as memory access patterns, have the potential to leak arbitrary information about the system. For example, a trace can capture the memory access behavior of a critical cryptographic function (which is known to be a function of the secret key [17]), a set of lookups corresponding to the parsing of a social security number, or even detailed system configuration parameters that are considered a trade secret. While the sharing of these traces between technology partners can lead to more robust and high performance systems, it can also leak highly sensitive information, and expose user data to security vulnerabilities.

It has been shown [18, 19] that safe ad-hoc anonymization is difficult to achieve. Given the cleverness of attackers working to undo well-intentioned, but ultimately insufficient, anonymization techniques [20], many have simply decided to cease making traces available altogether. Today when such traces are needed, programmers may be asked to "obfuscate" the key algorithm behaviors to hide sensitive data or provide "models" of the system which *approximate* the same behavior but omit sensitive parts. Hand-built "models" of the system are both tedious to code and of limited predictive power. Since there is no well-defined and well-trusted approach to this problem, developers are often forced to resort to rough human-language descriptions of the behavior of programs (e.g. "it is 80% pointer-chasing"). This leads to missed opportunities, frustrated optimization, and the design process ultimately suffers. Ideally, engineers would access methods to eliminate any sensitive information from the traces while still capturing the program behavior and its interaction with the underlying hardware. However, the extent to which "sensitive" data influences program behavior is rarely understood by a single party, and even harder to argue is that it is completely absent from a trace.

We present a new formulation of this problem where one knows *a priori* exactly how much information a trace is giving away in the worst case. The basic idea is to take a trace and squeeze it through as small a "hole" as possible to extract as much information as possible out of the trace without completely compromising the usefulness of the trace. Like *wringing* all of the water from a sponge, in the ideal case only the *structure* of the trace (the dry sponge) remains and all potentially sensitive data has been eliminated. While we have no mechanism of quantifying the amount of sensitive data that remains, we do have a way to say how much *total* information is provided, which yields a useful upper bound. In other words, while we cannot say for certain how much water remains in the sponge, we know that the amount of water has to be strictly less than the total volume we squeezed the sponge into. We observe that when compression is taken to this extreme and lossy form, it connects to security in this unexpected way. However, as is often the case in computer architecture, an important tradeoff remains between information leaked and degree to which the trace accurately captures the behavior across a suitable domain of possible options.

We formalize this new approach specifically in the context of memory address traces, as they are well studied and we have many prior techniques to build from. To explore the tradeoff exposed by this problem, we examine a new approach of performing guided memory trace synthesis building on ideas from signal processing. By projecting the address space onto a wrapped 2D image, we are able to decompose memory behavior into an orthogonal set of features that can then be replayed to reproduce the same "visible" patterns as the traces under examination. Specifically, we use a Hough-transformed version of the trace to find both constant and strided access patterns; Hough features are also used to concisely summarize the trace behaviors. Our contributions:

1. We introduce trace wringing, a new paradigm of anonymity and privacy in the context of traces where compression and modeling provide a way to release information with easily verifiable bounds on leakage.

2. We demonstrate a pipeline instantiating this idea in the context of address traces and show how signal processing techniques can be used to squeeze information out of traces while maintaining program behavior.

3. We verify through cache-simulation results that trace-wringing can be achieved as a proof-of-concept. While the resulting systems may still give away thousands or tens of thousands of bits, it opens the door to further optimization and refinement.

4. We compare our approach with prior work in address trace compression and synthetic trace generation. We are able to construct proxy traces using as few as tens of thousands of bits which is orders of magnitude fewer than compressed traces and the profile used in synthetic trace generation.

5. As a first evaluation of security beyond just bit leakage, we show that a class of existing AES attacks fails to find useful information in the traces processed in this way, which illustrates the utility of such an approach.

The rest of the chapter is laid out as follows. First, we present the new problem of "wringing" a trace more completely. In Section 3, we compare and contrast this problem to its related work on prediction, compression, and other classic trace analysis approaches. Section 4 describes our approach of using signal processing techniques for trace wringing. In Section 5, we describe our experimental setup, followed by an evaluation where we compare cache-simulation results. We summarize and conclude in Section 6.

2.2 Wringing a trace

A program trace can contain a tremendous amount of information about the system under evaluation. For example, memory accesses give away the data (e.g. secret keys) used in calculating the addresses, simultaneous accesses to different data storage areas can give away important relationships (e.g. between an individual's access rights and fields of a data structure they are accessing), and so on. But, as we know, such traces are invaluable for performance evaluation because they demonstrate the way the system actually behaves in the face of the workloads it must actually handle.

While the behaviors are important at a high level, rarely are the specific elements of the trace critical. Rather it is the relationship between those elements and the proportions that they appear in the trace that is often the key. This is of course not a new insight, and many people have attempted to capture these behaviors with microbenchmarks [21] and other trace synthesis schemes in the past [22]. What we claim as new is the idea that we can formalize these schemes in such a way that it *bounds* the amount of information leaked about a system being traced.

The argument is simple: if we only share n bits about a specific trace then we cannot leak more than n bits about that trace. In practice, this means that if we share only a few tens of thousands of bits of information about the trace, then nothing beyond

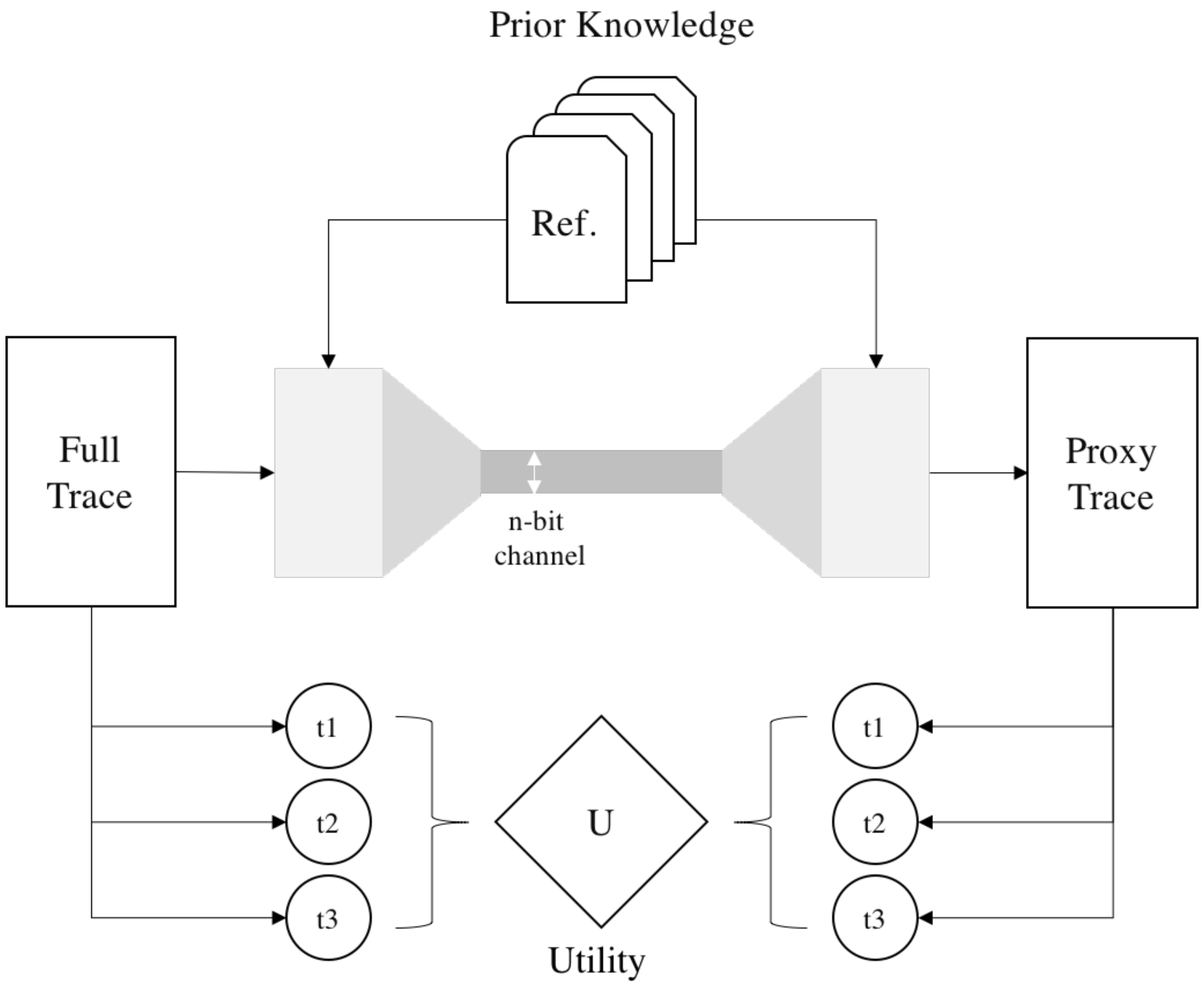


Figure 2.1: Forcing a trace through a channel with a capacity of only a few bits bounds the amount of sensitive data shared. While any public information such as prior non-private traces can be used in the creation of the code, the trace to be coded must not be known to the receiver. The objective then is to minimize the number of bits shared while maximizing the utility of the proxy trace. Here, we measure the utility in terms of whether or not certain tests *t1, t2,* and *t3* are passed by the proxy test and/or how close to the original tests results they get.

those bits has been leaked. While it is not a perfect solution (some information might be lost), it says something useful about the maximum amount of information that can be leaked. For example, it should be impossible to recover an extensive list of social security numbers, sensitive health information, or even an entire set of secret keys from such a trace. To maximize security one wants to give away as little data as possible about the trace. However, to maximize utility the opposite is true. Here is a new question for computer architects – how little can one give away from the trace while still being useful?

At first one might consider this to be exactly the problem of compression, and there definitely is a resemblance. Most compression schemes seek to perfectly replay a given input sequence by exploiting the fact that their inputs are far from completely random [23]. By understanding those common structures, for example the tendency for repeating patterns to occur [24], a more concise representation exploiting these structures is possible. Most modern compression algorithms start from a relatively blank slate and train a predictor of some form on the input as they process it. The duality between compression and prediction is pointed out by Chen et al. [25], who note that when you predict a value with high accuracy you can compress by storing an encoding that "the predictor is correct n times in a row" most of the time. Lossy compression is then a natural extension of this idea where the predictor is "close enough n times in a row".

However, even lossy compression schemes typically seek to minimize the error between the original trace values and the compressed trace values [26]. Here we have a problem that is different in two important aspects. First, while we want to keep the behavior of the trace to our tests the same, we may not care that the actual addresses themselves are similar. Second, we should be able to prime our scheme with data from other traces that do not contain a secret that we care about. In this way, we can think about this problem as attempting to decompose a trace into two aspects: a trace's "structure", and a trace's "data". The trace structure is what defines the hierarchy of patterns inherent to the trace that are useful for making statements about performance, while the trace data contains the specific set of addresses that makes the trace complete. The structure is all we really care to transmit and, when separated from the data, may be incredibly compact. The question then becomes, *how compact for how useful?*

Answering this question requires an analysis across two metrics: information and utility, as described in Figure 2.1. Information is surprisingly easy to quantify; it is the number of bits from the secret trace that need to be transmitted. Note that any number of bits about other traces or training data can be shared freely and even hard-coded into the receiver. Our approach is to describe traces as a probabilistic grammar of generators coupled with very high level accounting of behavior over time and account for bits in both the structure and parameters of this scheme. Quantifying utility is harder and more use-case specific. We define a distance function between cache miss-rates of trace vectors as one such function, but understand there are many other metrics one might use [27, 28, 22].

While this problem is generalizable, we are considering address traces for this initial class of experiments. While many other classes of traces might benefit, address traces are some of the most well studied and understood, and provide the most stable foundation for this new work to be developed upon and evaluated.

2.3 Related work

In this work, we start with a security parameter (the number of bits we tolerate giving away) and analyze a program's behavior by studying its address trace to eliminate information that is not essential to describe its behavior down to that security parameter. At the heart of it, we want to accurately characterize a program's trace, and preserve only the bare minimum information, so as to not leak it unintentionally. This new problem can then leverage much of the related problems in the fields of trace compression, statistical program profiling, synthetic trace and benchmark generation, and data privacy and anonymity. In the rest of this section, we will compare and contrast our work with the large body of work that precedes it.

2.3.1 Trace compression and approximation

Trace compression is well studied. TCgen [29] has a compression ratio as high as $77,000$ for certain benchmarks. Lossless algorithms exploit sequentiality and spatiality, value prediction [30, 23, 31], perform loop detection and reduction [24], convert absolute values to offsets [32], and use clustering to improve compression [33]. ATC [26], a compression tool for cache-filtered addresses, is capable of both lossless (using bytesort) and lossy compression (using sorted byte-histograms).

Compressed compact representations are used to understand and predict program behavior. Larus's work on whole program paths [34] introduces a method to determine a program's dynamic control flow, using the SEQUITUR [35] compression algorithm. Chilimbi presents a similar scheme to effectively represent a program's dynamic data reference behavior [36], also using SEQUITUR. Trace Approximation [37] generates compact summaries of memory accesses of parallel applications to achieve trace reduction.

2.3.2 Characterizing program behavior

Eeckhout et al., have described a method to obtain detailed statistical profiles within program traces [38] with the combination of microarchitecture-dependent and -independent profiling tools. Their syntactically correct, and representative synthetic traces can be simulated on existing simulation tools. Machine learning algorithms are to understand large scale program behavior by clustering basic block vectors to find the representative sections of a program [39].

Chen et al., have shown that hardware event profiles for feedback-directed optimizations, can be improved by using machine learning and statistical techniques[40]. Oskin et al. collect statistics from actual program simulation to generate a synthetic benchmark [41] that is faster to run. While statistical methods are useful in modeling behaviors of programs, they do not consider the amount of information they inadvertently leak. It is worth revisiting these works in the context of how much total information they leak versus how useful they are across a range of optimizations. We leave unifying these approaches in the context of wringing as future work.

2.3.3 Synthetic trace generation

Synthetic trace generation has been a classic solution to characterize performance and effectiveness of novel designs (when workloads do not exist) [42]. To ensure that the synthetic traces behave as expected, Thiebaut et al. adhere to a hyperbolic probability law [43, 42]. Other methods on artificial workload generation have been described [44] and reviewed [45]. PSnAP [46] separates the program structure from the memory access pattern in two phases: capture, when PSnAP generates a profile using PMaCInst [47], and replay, when it produces a synthetic trace based on the captured profile.

For HPC applications, Weinberg et al. determine memory signatures and mimic them to generate synthetic traces [48]. They maintain the cache miss rates of the applications under test with Chameleon [22], a memory locality analysis tool suite. The tool produces a small seed, which is replicated to construct an arbitrarily long trace. BenchMaker [21] is a parameterizable and scalable synthetic benchmark generator, which can create customized workloads given some (forty) microarchitecture-independent program characteristics.

Unlike the previously discussed papers, BenchMaker creates *benchmarks* which can then be run on real-hardware (or simulators) in order to better explore the application space. Van Ertvelde et al. go further and propose code mutation [49] for generating benchmarks that hide functional semantics of proprietary programs. They do this at the binary level of chosen benchmarks rather than on traces.

2.3.4 Preserving data privacy

Differential privacy [50] protects anonymity by adding some amount of carefully calibrated noise to the sensitive data sets so as to maintain the main properties under study. Access to the system is metered out carefully to ensure privacy is maintained while being as true to the original distribution as possible. It has been pointed out recently [51], that differential privacy may introduce an unacceptable amount of error. *Being able to add noise to address traces in this fashion may not result in similar or expected program characteristics.*

Plausible deniability [52] presents a formal framework to generate synthetic data records efficiently while guaranteeing privacy. Their data synthesizer is based on a probabilistic model; it captures the joint distribution of attributes collected from the real dataset. Their target applications include machine learning and dataset analyses. Other formalizations of privacy are an active area of exploration with k-anonymity [53], i-diversity [54], t-closeness [55], and many others.

Traces are inherently time-series data sets. They map less clearly onto these models where a set of queries are often asked and answered by someone with the full data set. Unifying trace analysis and these models of privacy appears to be an open problem and our work stands out from the ones described here both by its intent and simplicity. We provide an up-front security parameter, the total amount of bits to be leaked, and we squeeze our traces to that level. This approach provides a useful point of comparison as more advanced techniques linked directly to more specific security models are developed and evaluated. Drawing inspiration from information theory, we also try to find an upper-bound on the information leaked from the system by trying to quantify the number of bits of information given away by our method while trying to minimize it.

Another related field is quantitative information flow analysis; similar to differential privacy it proposes numeric measurements that pertain to privacy. Some examples of its applications are in producing better bug reports which maintain user privacy [56] and measuring source-location information leakage in wireless sensor networks [57] among many others. McCamant et al., present a method to determine how much information real programs leak [58] using a practical implementation of quantitative information flow which uses dynamic analysis.

2.4 Our approach to wringing

Traces expose the inner workings of a program, its interaction with the runtime, and the underlying hardware architecture. As such, even the simplest memory traces prove to be a complex concoction of patterns generated by these underlying factors. For example, in a memory address trace, accesses to many different types of objects across both stack and heap are all interleaved to create the whole. Our goal of capturing the *structure* of these traces first requires that we identify, describe, and quantify the patterns that we care most about. While understanding the underlying cause of these patterns requires detailed knowledge of the program, quantifying the magnitude of these patterns can be done on the traces alone. In fact, it is observed that even complicated programs exhibit memory access patterns that can be decomposed into simpler ones.

To get a visual sense for the structure of such traces, we project the address trace

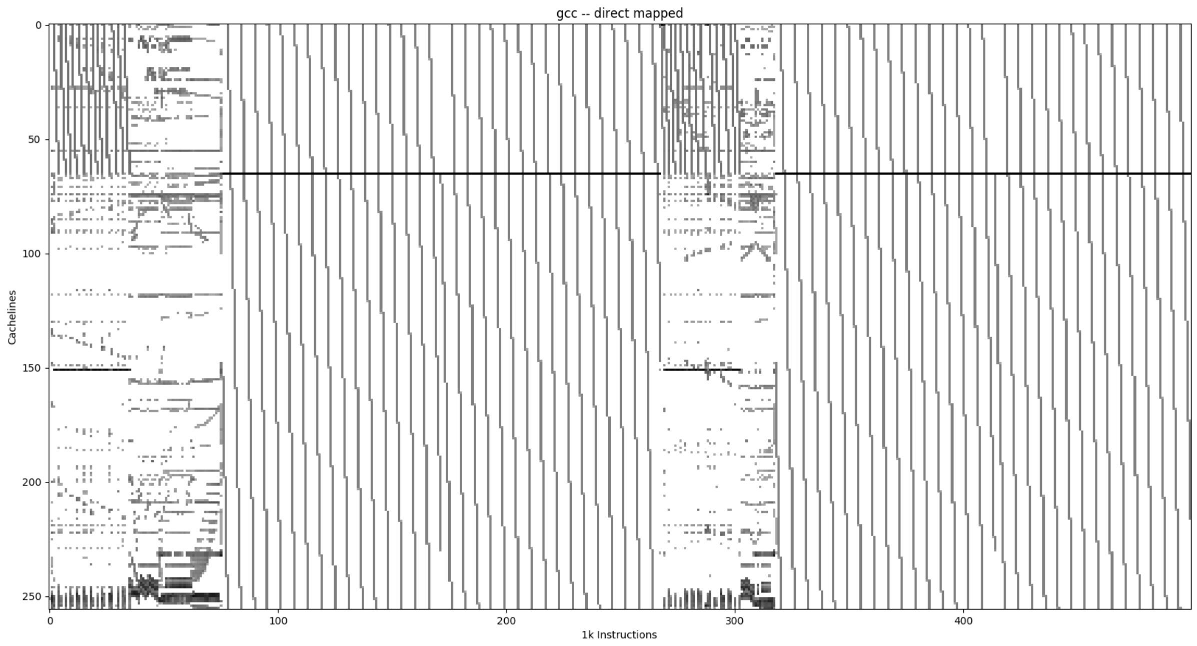


Figure 2.2: The modulo-memory access heatmap for `gcc`. The heatmap is an $N \times M$ sized graph, where N is some high power of 2 and M is the number of 10000 instruction windows in the trace. These *modulo-memory access heatmaps* illustrate patterns that exist within program executions, and give us a visual sense of memory access activity. When mapping longer traces, for example, we see phases (as in 2.4), but we also observe local patterns within these phases as shown here.

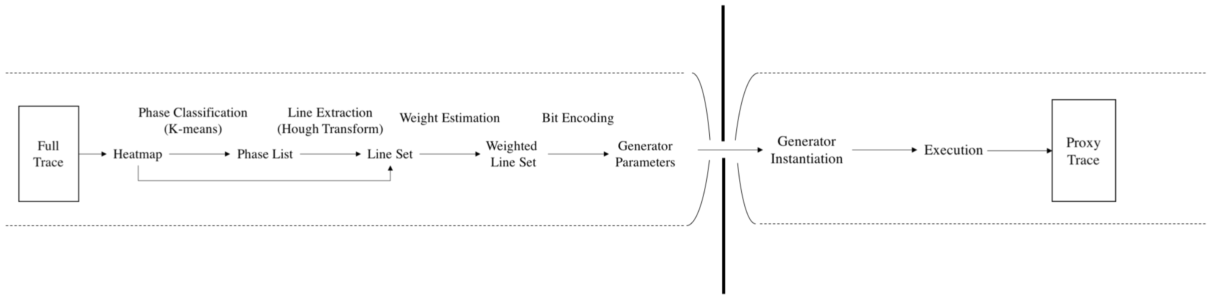


Figure 2.3: Pipeline for our approach to trace-wringing for proxy trace generation. The problem of sharing information can be described with two subsystems; at the trace-wringing end, we find parameters that will accurately generate the trace at the generator subsystem end. The goal is to minimize the size of the packets being sent between the two subsystems, while still maintaining integrity of the data transmitted.

onto a fixed-size modulo-mapping of the memory space. This *heatmap* is a graphical representation of the memory access behavior over time. Figure 2.2 shows such a heatmap for `gcc` where instruction count (time) runs along the x-axis and the address runs along the y-axis. If we were to plot this for the *entire* memory it would clearly be too large for such a graph (the distance between the stack and heap would dwarf any local behavior), so we instead plot the address modulo a large power of two. We call that the "wrapped address". This plot of the wrapped address over time (in terms of instructions) has the advantage of mapping addresses onto a more manageable space, but at the same time keeps the spatial-temporal structures that would actually impact a real cache. The *darkness* of each pixel is a function of the total number of memory accesses that happen to that wrapped address during a window of instructions.

Interesting and intuitive patterns emerge after looking over this graph. The flat horizontal lines in the graph are patterns of repeating access to a set of addresses. These are high temporal locality behaviors. Sharp diagonal lines, on the other hand, are regions of high spatial locality as addresses are accessed one after the other in succession. If we can concisely capture the *character* of these behaviors, without transmitting the addresses themselves, we can minimize the amount of information leaked. Describing an efficient method for extracting these patterns is exactly the goal of this section.

Figure 2.3 gives a high-level overview of the pipeline we propose to first wring and then expand a trace. There are two essential subsystems in our pipeline; one for extracting structural information about the trace from our heatmaps, i.e., for trace-wringing, and the other for rebuilding a proxy trace with the same structural information. At one end, as seen in Figure 2.1, with the help of some prior reference knowledge about traces, a full trace is decomposed into its describing parameters. These parameters are the ones being communicated via a constrained channel to the generator subsystem, which then uses the same prior reference knowledge and the descriptive parameters to generate a

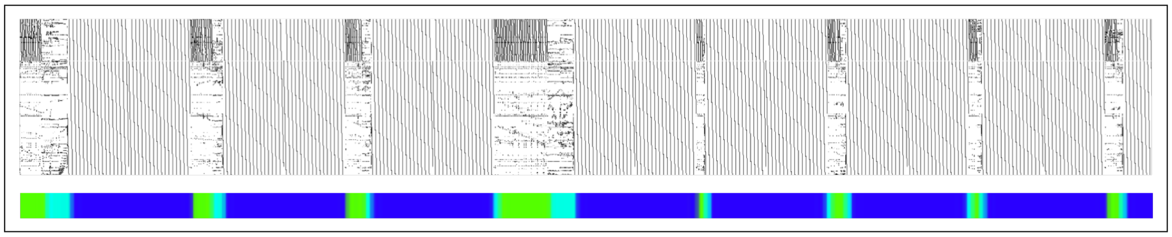


Figure 2.4: Phases visible in the trace generated by `gcc` after k-means clustering. Each of the 3 colors in the bottom marks a unique phase in the trace. Note, importantly, that phases reoccur over time.

proxy trace. In our pipeline, prior reference is used for optimization of encoding (generation of heatmaps, detection of phases and line segments within them, and creation of "information packets"), decoding (proxy trace generation from shared "information packets"), and the selection of Hough parameters. The generated proxy trace's utility is measured by testing its properties against that of the original full trace.

The modulo-memory heatmaps exhibit hierarchical organization. Globally, there exists a recurrence of similar patterns in the order of a few tens of thousand instructions, i.e., the presence of program phases, and within them, we observe patterns that we associate with the more local memory access activity. In order to find some representative of the higher echelons of this hierarchy, we employ k-means clustering to detect the program phases [39].

2.4.1 Phase detection

While Figure 2.2 is not the full execution of `gcc`, we note the presence of a set of program phases. The first observation we make is that if we wish to capture the character of these traces, we need to extract higher level shifts in behavior over time. If one can group together alike behaviors (for example, the middle and end of Figure 2.2) we can then select only a *single representative* for each such behavior. Fortunately this is almost exactly the problem of phase detection [59, 60, 61]. To find the phases, and select a representative, we pose this as a clustering problem (similar to prior work). We break the execution up into a set of "chunks" by instructions executed. The columns of the chunks are then summed together to form a vector. Each vector thus has a length equal to the number N of wrapped line addresses. We can think of each of these vectors then as a point in N dimensional space. Finding groups of similar points (our memory vectors) is then exactly the clustering problem. Here we can simply apply the k-means algorithm [62] with k equal to the number of phases we wish to represent in the trace. The k-means algorithm represents clusters by a set of k cluster centroids which it then iteratively optimizes. Each iteration alternates between assigning each point in the space to exactly one centroid, and updates centroid position to be in the "middle" of the new set. After k-means, we take each cluster and select one that is the longest to be the representative cluster.

Figure 2.4 shows the result of running the phase detector on the memory address trace for `gcc`. Each of the 3 colors labels the trace above it with a unique phase identifier. The technique does a good job of lining up with the repeating structures.

Now, with these phases marked, rather than encoding the full trace monolithically, we can encode just the k representative clusters independently with $log_2 k$ bits. The list of the phase identifiers can then become part of the information shared. As can be seen in Figure 2.4, there is a great deal of temporal locality in the phases and can be trivially compressed by another order of magnitude with run-length encoding.

Given that we now have a set of representative chunks of execution, we need to efficiently summarize the features that exist within each chunk. If we look back to Figure 2.2, we can see that many of the patterns in the heatmap can, in fact, be reduced mostly to a set of lines.

2.4.2 Decomposing with Hough transforms

Concisely summarizing all of the complex patterns of the trace all at once can be overwhelming. However, if we can break the pattern down into a set of simpler behaviors, we can then tackle them one by one. Given that both strong temporal and spatial locality features show up as lines, decomposition into a set of line segments is a natural place to start. However, decomposing the address trace features in the space of $\textit{wrapped\_addresses} \times \textit{instruction\_count}$ directly is not easy. Luckily, we can draw upon established methods in image processing to transform our heatmaps into a space where such extractions are achievable.

The Hough transform [63] is a popular computer vision procedure used to detect patterns in images. The technique is used to find the locations and orientations of certain geometric primitives in the given space. Hough transforms, being resilient to noisy images, makes for an ideal feature extraction candidate for our problem. Geometric primitives such as lines, ellipses, and circles are supported by Hough transforms, but we find use only for the simplest Hough transform: the Hough-line Transform.

While standard regression methods are useful fitting a slope-intercept form of $y = mx + b$ to a set of points, finding *sets* of rotated lines from an image is hard in the Cartesian coordinate system. The Hough-line transform employs the polar coordinate form and describes lines by their distance from the origin r and the angle formed between the origin and the closest point on the line θ: $r = x \cos\theta + y \sin\theta$.

Now, we have two separate coordinate systems in which we can find the best fit line; the image space, and the ¡r, θ¿ parameter space. For every point in the image space, the Hough transform considers every possible rotation of lines passing through that point. Iterating through the different possible values of r and θ in the Hough space, the algorithm forms a sinusoidal curve for each point in the image space. Each point

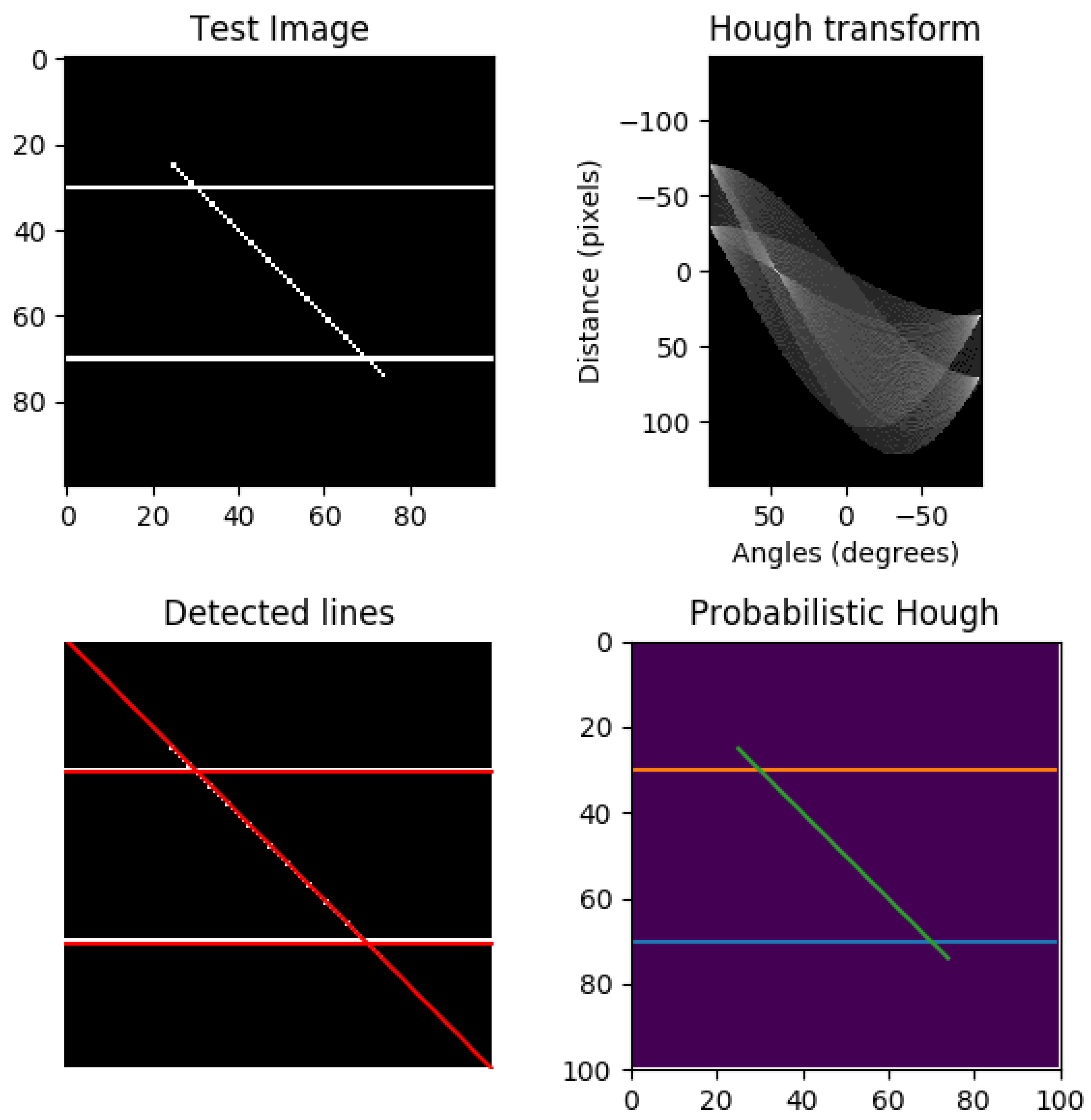


Figure 2.5: We capture information about lines we observe in trace heatmaps using the Hough Transform. Here, we demonstrate its working. The points on the test image are surveyed for parameters in the polar coordinate space described as the Hough Transform. The intersections describe the parameters of the detected lines. The final figure shows the Probabilistic Hough Lines, the more robust and efficient algorithm. For our heatmaps, we use the Probabilistic Hough Line algorithm.

in the ¡r, θ¿ space corresponds back to one possible straight line in the image space. This point-to-curve transformation (where every point in the image space is a curve in ¡r, θ¿ space) is the Hough-line transform. We do this for all the points, and the most coincident points (where the most sine curves intersect) in the ¡r, θ¿ space is the choice of parameters for a line in the image space. Specifically, what makes the Hough transform robust is how the parameter space is set up: it is divided into a mesh of finite intervals or accumulator cells. As the algorithm proceeds from point-to-point in the (x, y) (image) space, the accumulators in the discretized ¡r, θ¿ space are incremented.

For our instance, we use the progressive probabilistic Hough transform [64], a rendition of the Hough transform algorithm that only performs voting on a subset of the input points. These input points are chosen based on certain features of the expected result, such as a threshold of "darkness", the length of the expected line, interpolation strategies, and the angle of the line. By interleaving the voting process with line detection, this algorithm finds the most prevalent features first, while also minimizing the computational load.

The progressive probabilistic Hough transform returns a set of lines, with each line's (x, y) coordinates in the modulo-memory heatmap space. We also introduce a variable, "weight", for each line, which is a measure of darkness of the line.

The list of phase identifiers (the result of clustering), the two (x, y) coordinates of each line detected by the Hough transformation, and the line's weight per representative phase, give us the amount of share-able information.

2.4.3 Proxy trace generation

Using phase detection and Hough-line transformation, we end up with a set of Hough lines for each representative phase. Each phase is also assigned a label indicating to which cluster it belongs to, i.e., which representative phase "represents" it. Since the structural information of each phase is encoded in the the Hough lines, we can generate an "address tracelet" for each phase using the representative's Hough lines.

Phases from the same cluster may occur intermittently and in different lengths. For all phases in the same cluster, we generate patterns continuously in a rotating fashion regardless of the length. For example, if phases x_1 and x_2 are both represented by representative phase r_1 (suppose x_1 occurs before x_2 and there's no other phases represented by r_1 in between), we then generate a trace for x_2 following the partial patterns we generate for x_1 and wrap over if the total length grows beyond r_1, i.e., the starting time step t when generating addresses for x_2 will follow the end time step $t - 1$ when we generate for x_1 and wraps over when t becomes larger than the end time stamp in r_1.

Within each phase, we generate addresses by alternatively picking addresses from the subset of lines that cover each point in time (each time step t in the projected address space corresponds to N addresses, in which N is determined by the window size when the heatmap is generated at first place). If there are no lines covering the current time step t, we generate addresses for t from a uniformly distributed noise function as there is no clear pattern observed by the Hough transformation and we mimic a random access behavior in this way.

Upon picking a Hough line at time t, we generate an address "segment" from that line based on a fixed segment length, which captures locality at a small granularity. The segment length for each workload is hand-picked so that it best captures characteristics of the trace. Each address generated from the line is also shifted to the left by the cache block offset bits (6 bits for a typical 64B line size) since the purpose of wringing is to preserve the cache-level patterns.

After generating address tracelets for all the phases, we concatenate them together in the original order of the phase occurrences to form a complete proxy address trace. The proxy trace has the same length as the original trace but its memory footprint is limited to the wrapped address space.

2.5 Evaluation

To evaluate the effectiveness of the approach, we take a set of traces, wring them through our pipeline to a target number of bits, and evaluate the traces across a range of cache configurations with regards to miss rate. The details of the parameters and process follow below.

Starting with the full traces, we first convert them into heatmaps which are parameterized by the number of instructions from the trace to simulate, the window size, and the total size of the mapped space. If a map space is chosen to be too large, the line detection techniques will fail to pick up useful edges as there is too much white space for them to operate properly. If the map space is too small then the addresses will be truncated to such a degree that they will cease to be useful for evaluating miss rate. For our experiments, the x axis in the modulo-memory heatmap represents 10,000 instructions.

We use signal processing techniques here to collect important information about the heatmaps. We compute the Hough transforms, as described prior, to give us the value of the constants that describe the lines that the algorithm is able to "see" in the heatmaps. Specifically we must hand-tune the progressive probabilistic Hough transform input points (to reduce the search space of the algorithm) to find the lines in the midst of all the noise that these heatmaps inherently have. For our experiments, the parameter *threshold* ranged from [20,200], *line_length* ranged between [10,60], *line_gap* ranged between [1,50], and *theta* ranged between π and $\pi/2$. Specifically, the probabilistic Hough lines [65] are then generated and remapped back into the address space.

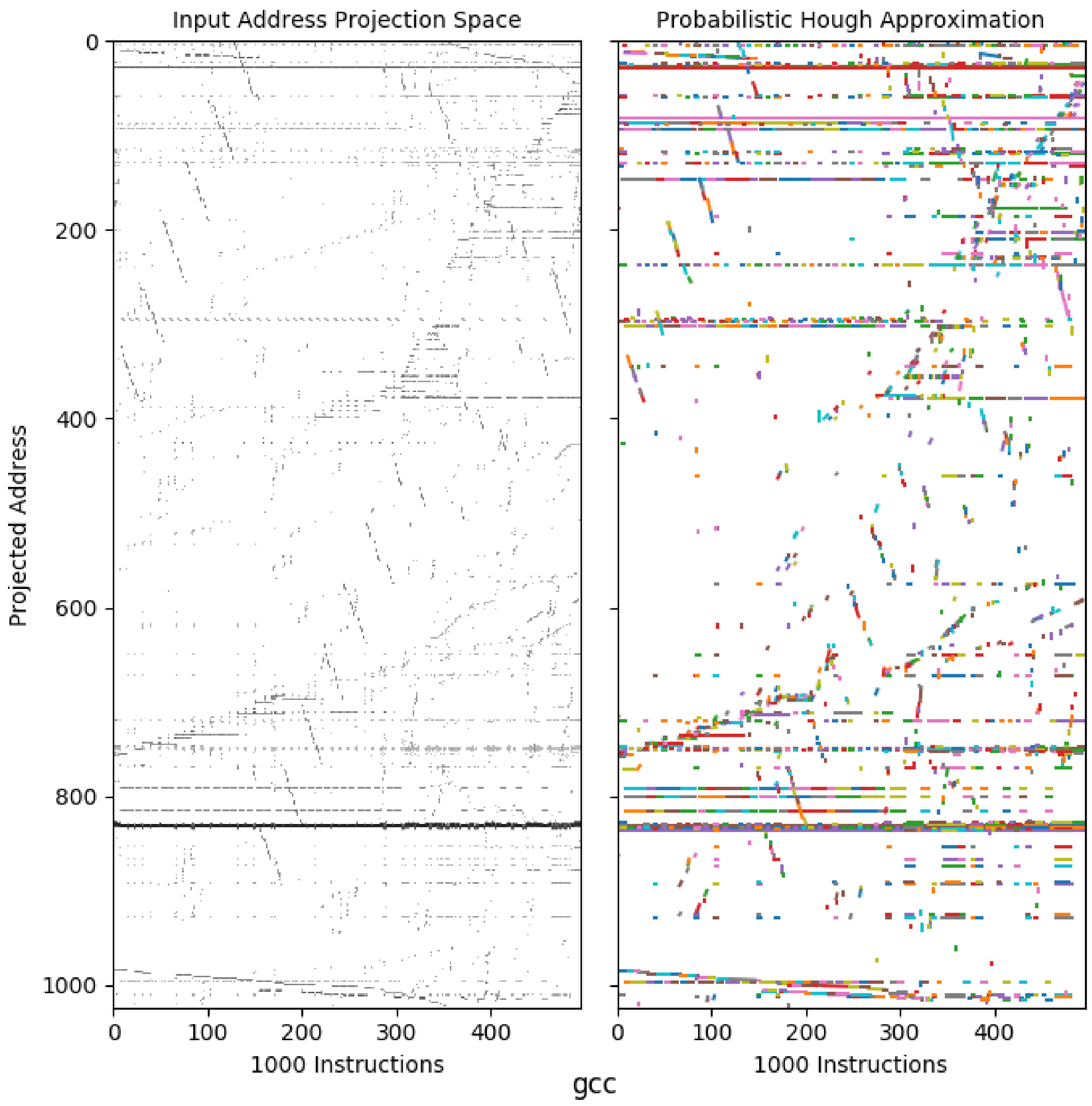


Figure 2.6: Producing probabilistic Hough lines on top of the heatmap of the SPEC2006 benchmark, `gcc`. The colors are used to indicate distinct lines produced by the decomposition.

2.5.1 Measuring bits

While our main goal so far has been to extract and describe the structure of traces as correctly as possible, we must also maintain that not too much information is given away. The information that needs to be transmitted to the trace generator must contain both the global phase-identifier information, and the line coordinates and weights per representative phase.

$$Phase\_bits = \lceil log_2(\#\_phases) * len(phase\_seq) \rceil \tag{2.1}$$

To calculate the bits that are needed to produce the proxy trace for each workload, we dump all the labels from the clustering result as well as all the Hough lines detected, each of which is a 5 tuple of coordinates in the heatmap space and a weight value. The phase information can be represented using *Phase_bits* (Eq. 2.1). We then apply a variety of compression techniques to compress the dumped files and estimate the bits of information by measuring the size of the compressed file. We push all of the information that is to be measured into a single file to ensure that no side information is accidentally shared between the two halves of the system. We discuss the breakdown effects of each compression technique in Section 2.5.4.

2.5.2 Trace selection

Rather than working on the traces in their entirety, for each workload, we evaluate from a large SimPoint [39] trace of the most representative region of 100M instructions, which results in a variable length of address traces from 30M to 70M accesses for different workloads. We use benchmark subsetting suggestions [66] to reduce the space of evaluation to a more manageable level, although our results are limited to 6 of the 9 suggested due to errors getting the benchmarks running. Results from all benchmarks run are considered and the optimal (in terms of bits leaked and accuracy of miss rate) points at two different levels of bit transmission budget are shown in Table 2.1. The time overhead for our pipeline is also presented in Table 2.1. Although it varies between different workloads, we expect this overhead to grow sub-linearly as the trace becomes longer for any single workload. The time overhead is linearly correlated with the number of distinctive phases in the trace and the number of phases tends to grow very slowly since phases often repeat themselves.

2.5.3 Measuring utility

As we concentrate on cache behavior as a target for initial evaluation we use cache miss rates pre-wringing and post-wringing to evaluate how useful the resulting trace is. The collected address traces are simulated with different cache configurations using DineroIV [67]. We use 6 cache configurations in our experiments: direct-mapped and 4-way associative combined with 3 different cache sizes (8k, 16k and 32k), and measure their miss rates.

From Table 2.1, we observe that as the bits of information leakage increase, the miss rate gets closer to the ground truth miss rate, which confirms that, with more information going through the wringing "hole", the proxy trace we reconstruct becomes more similar to the original trace in terms of structure. Some benchmarks such as *sjeng* and *hmmer* do not benefit much from the extra bits, in terms of closeness to the miss rate, as $10,000$ or even fewer bits are *enough* to accurately capture their cache behavior, while others including *libquantum* perform much better due to the fact that they have a more complex structure which requires more bits to encode.

Figure 2.7 compares the proxy heatmap generated for gcc against the original. Our wrapped address space is of height 2048 (lines in the heatmap) and each "column" in

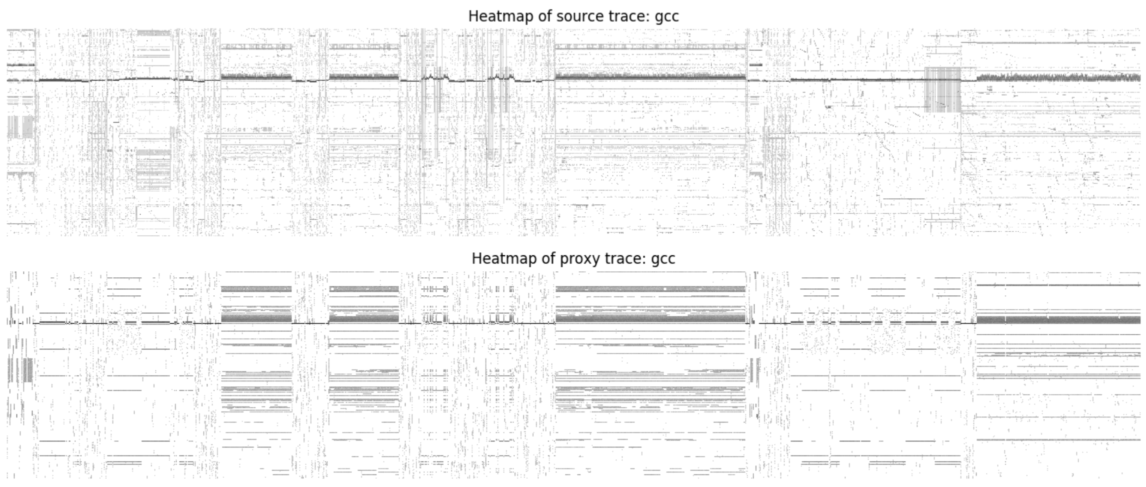


Figure 2.7: Heatmap for the original `gcc` trace and the trace-wrung proxy generated for `gcc` trace from the wrapped address space. Each pixel corresponds to one wrapped address at one time step. The darker the pixel, the more times that address is accessed during that time step.

the heatmap corresponds to 10,000 memory accesses. The figure illustrates that our approach is able to capture all but the subtlest patterns.

2.5.4 Comparison to existing compression and trace generation techniques

We are not aware of any prior methods that have attempted to bound the information leakage from generated traces. While our approach to bounding draws from trace compression and synthetic trace generation techniques, we stand out in at least the following ways: (a) we seek similar *behavior* in our generated traces, rather than similar addresses, (b) we allow unbounded priors from non-sensitive traces, (c) our traces are lossy specifically in a way that it maintains architectural utility, and (d) qualitatively, the target size of the final “compressed” trace is far smaller than normally considered. This last point, (d), is something that we can quantify experimentally.

Table 2.1: Best miss rates observed for the benchmarks with three different bit-budgets of information leakage and time overhead for trace-wringing followed by proxy trace generation. For each cache configuration 4 miss rates are reported. We report: ground truth miss rate from the original trace, best miss rate using *all* hough lines, best miss-rate with 100k bits, and best miss-rate with merely 10k bits. “-” means the most aggressive setting in our experiments requires more bits to construct the proxy traces.

Benchmark	Bit Budget	Cache Configs.						Time	
		8k,dm	8k,4w	16k,dm	16k,4w	32k,dm	32k,4w	Wringing	Decompression
gcc	Orig.	6.88%	3.91%	4.86%	2.79%	3.36%	2.11%	138.55s	123.37s
	Full	6.10%	3.98%	3.60%	1.27%	1.93%	0.48%		
	100k	4.82%	2.94%	2.81%	0.72%	1.40%	0.25%		
	10k	-	-	-	-	-	-		
sjeng	Orig.	12.3%	5.01%	6.45%	2.19%	4.24%	0.64%	94.42s	128.08s
	Full	12.85%	10.16%	8.22%	3.74%	4.26%	0.64%		
	100k	12.85%	10.16%	8.22%	3.74%	4.26%	0.64%		
	10k	11.89%	7.78%	1.13%	4.39%	0.25%	2.25%		
cactusADM	Orig.	8.29%	7.03%	5.44%	5.29%	2.09%	1.54%	209.94s	918.04s
	Full	9.35%	4.98%	5.21%	0.85%	2.08%	0.29%		
	100k	3.73%	0.49%	2.02%	0.14%	0.55%	0.12%		
	10k	-	-	-	-	-	-		
milc	Orig.	7.99%	7.09%	7.68%	7.03%	7.35%	6.94%	336.41s	31.36s
	Full	7.73%	7.19%	7.11%	6.66%	5.93%	5.69%		
	100k	7.51%	7.25%	6.75%	6.44%	5.46%	5.44%		
	10k	-	-	-	-	-	-		
hmmer	Orig.	27.8%	2.54%	26.8%	1.20%	17.0%	0.78%	151.79s	287.95s
	Full	23.6%	7.21%	20.53%	5.05%	10.31%	4.32%		
	100k	23.6%	7.21%	20.53%	5.05%	10.31%	4.32%		
	10k	23.6%	7.21%	20.53%	5.05%	10.31%	4.32%		
libquantum	Orig.	16.3%	16.2%	16.2%	16.2%	16.2%	16.2%	57.73s	21.89s
	Full	17.31%	17.27%	14.99%	14.90%	12.10%	11.90%		
	100k	17.31%	17.27%	14.99%	14.90%	12.10%	11.90%		
	10k	74.46%	74.44%	69.33%	69.31%	59.31%	59.32%		

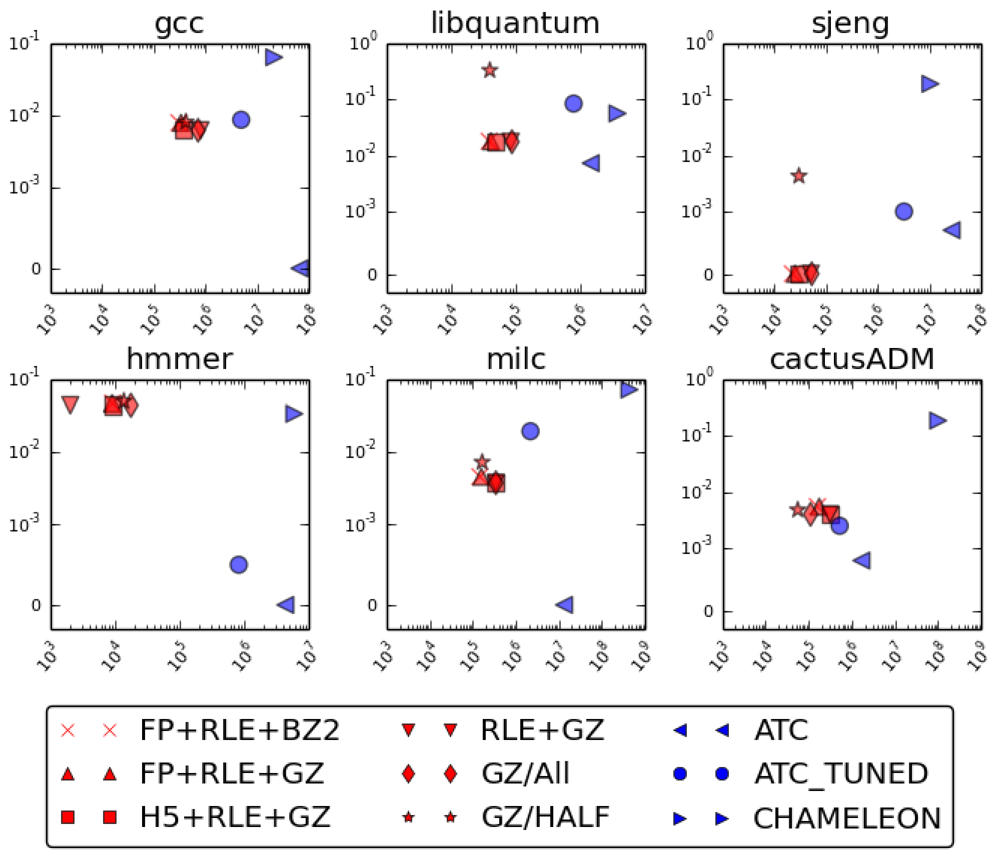


Figure 2.8: Breakdown of trace-wringing pipelines and comparison against state-of-the-art compression and synthetic trace generation techniques in the bit-error space. The x-axis represents *number of bits* transmitted, y-axis represents the *geometric-mean* of error in miss rate. Per workload, we mark the bit-error points for different techniques; being in the lower-left is better. A packet contains information about hough lines and labels. "FP" is fixed-point quantization on hough lines, "RLE" is run-length encoding on labels, "H5" is the HDF5 format compressed using h5py [68] for hough lines. We use a general purpose compressor on our packets, either Gzip,"GZ", or Bzip2, "BZ2". "GZ/ALL" and "GZ/HALF" indicate Gzip on unquantized packets of either all or highly-weighted half of the hough lines. "ATC" is the off-the-shelf lossy compression [26], "ATC_TUNED" is hand-tuned to minimize information transferred. "CHAMELEON' is from the open source implementation of Chameleon [22]

Specifically, we compare our method against a state-of-the-art lossy compression and synthetic trace generation in Figure 2.8. “ATC” is an open-source implementation of the address trace compression framework [26], which supports lossy compression over cache traces. We run both off-the-shelf ATC, and a hand-tuned version that attempts to further minimize the trace size while still decompressing into useful traces. Although off-the-shelf ATC achieves good accuracy, it requires up to tens of millions of bits to represent the structure and data of the original trace in most cases. Even the hand-tuned version, which adjusts the similarity threshold and reduces the size of the unit of comparison, does not change the result significantly. This is orders of magnitude more than the number of bits transmitted in our trace-wringing framework (note the base 10 log scale). For synthetic trace generation, we use an open-source implementation of the Chameleon framework [48]. The profiles/characterization of traces are quite large even after h5 compression due to the fact that a histogram of address reuse is entirely captured in order to generate a similar-behaving synthetic trace. “FP+RLE+BZ2”, our most aggressive post-wringing compression technique, significantly reduces the number of bits while maintaining good accuracy. This is not to say that these and related approaches could never be improved to be competitive on this new problem, but both out of the box and with some careful tuning, they do not appear to be currently.

2.5.5 Case study: AES attack

While it is impossible to say with certainty what could be leaked in the resulting bits, it is worthwhile to examine the technique practically in the context of a known attack. Specifically, we choose to examine the trace to see if it is possible to recover an AES key using known attacks. AES attacks based on cache sets have been well-studied [17]; we follow a similar process here.

The vulnerable portion of an AES trace lies in the accesses to the Rijndael substitution function (`sbox`). This is stored as a table in memory. In the first round of encryption, the offset into the table is the result of each byte of the key xor'd with each byte of the plaintext. When the attacker chooses or knows the plaintext, the offsets are of obvious importance — the ability to discover the table offsets directly leads to discovery of the secret key. Because the post-wringing trace consists of cache set indices, we limit the attack on the original trace to cache sets only as well for a fair comparison.

The attack model is as follows. Assume the attacker has chosen a uniformly random plaintext, and made N calls to an AES encryption, where each call has 16 bytes of the plaintext. The attacker can observe the resulting traces, either pre- or post-wringing. The attacker prepares a table of 256 "candidate" values for each byte of the key. Then, for each key byte, the attacker considers every address in the traces that could potentially fall within the `sbox` table. Each of these addresses corresponds to an `sbox` table offset, and, when xor'd with the appropriate plaintext byte, yield a candidate key byte. The corresponding entry of the candidate table is incremented by one. When finished, the key byte with the highest candidate score is used in the key guess.

The vast majority of addresses processed will not be `sbox` accesses; however, because the plaintext is chosen to be random, these will become uniform random noise. Only the first-round `sbox` accesses always come out to the same value when xor'd with the random plaintext: the correct key byte. With enough traces, the signal corresponding to the correct key will rise above the noise and be readily apparent. In our attack, looking at full addresses, it took only 13 encryptions to get all bits of the correct 16-byte key.

Since the post-wringing trace is a smaller space of bits, we are unable to attack full addresses. Instead, we attack the bits provided; this makes the attack very similar to the original cache attack [17]. Attacking the first round of AES cannot yield all the bits of each byte of the key, since the offset within a given cache set is unknown. Attacking subsequent rounds of AES can provide the rest of the bits, but requires that the first round attack is successful. Therefore, showing that the attacker is unable to succeed in attacking the first round is sufficient to demonstrate that the attack fails.

We perform this attack on a set of traces collected from runs of Tiny AES [69] with a random plaintext. We perform the same attack pre- and post-wringing. In the pre-wringing trace, we use only 12 bits of the address (the amount of information contained in the post-wringing trace), masking the lower three bits and the upper bits of the address. We note that this trace was wrung with 8-byte cache lines specifically to give advantage to the attacker and show the usefulness of the approach; increasing the cache line size only makes the attack more difficult. Pre-wringing, the attacker correctly guesses the upper five bits of all 16 key-bytes after 1,838 encryptions. This is the maximal information that can be learned in a first-round attack with 8-byte cache lines. Post-wringing, the attack guesses wrong for all 16 bytes of the key after 50,000 traces.

We performed an entropy calculation on the original traces based on the distribution of addresses at each time step across a number of traces. We see that $\sim$160 addresses have more than $5x$ the information content of the remaining addresses. These higher information-content addresses correspond to the `sbox` computations. Post wringing, all addresses have uniform information content, i.e., there is no set of addresses that is more influenced by the key than others.

Our wringing process was able to produce a new trace with comparable cache miss rates. We received 0.0% (new trace) against 0.9% (original trace) for the direct mapped cache and 0% (both new trace and original trace) on the 4-way associative caches while completely stopping our AES cache attack.

2.6 Conclusion

The conflict between the need to share information (to provide more optimal performance) and hide information (for privacy) is becoming increasingly fundamental in the computer system fields. While addresses are one such type of trace, one can certainly understand how related problems exist with storage traces, cache coherence traffic, energy usage, user interaction data, and certainly location data. Clever, yet complex, techniques have been developed to address certain anonymity problems in the past, yet the reality is that they are often dependent on specific assumptions such as a lack of prior information, statistical distributions governing the data, or that number of queries can be tightly bounded. While our wringing approach is very direct, that directness also comes with clarity as to what it does and does not do. It does not *guarantee* anything about how useful the resulting trace will really be for optimization. However, it *does* transform the problem of safe sharing into a *measurable* systems problem subject to the myriad tools we have at disposal for *common-case optimization*. Furthermore, it *does* provide a *strong and clear bound* on the amount of useful information given by the trace.

The technique we present here is a proof-of-concept and we make no claims that it captures anywhere near the true minimum leakage to utility tradeoff. There is much work left to be done to bring the number of bits shared compared to the accuracy lost down into a more appealing tradeoff. $10,000$ bits, let alone $100,000$ bits, is still a tremendous amount of information to leak and it is far from certain that it can never be used for anything malicious. From a security standpoint, we must do far better than that. Despite this gap, we feel that even these results are better than the other approaches, which fall to the extreme of either leaking almost no information with limited connection to reality or direct connection to observed behavior and completely unbounded information sharing. We establish this experimentally in Section 2.5 by comparing against existing approaches, which while designed for different purposes, do functionally provide a bit-reduced trace with diminished fidelity. The specific set of techniques we propose push the traces to much lower levels of leakage than these other past works can achieve with only slight losses in accuracy. This is perhaps not surprising as the levels of "compression" one needs to achieve to store a trace efficiently on disk are far less than that needed to have confidence there is little sensitive information retained.

Looking forward, with this new approach we can build on years of community experience dealing with address traces and encode common patterns in a general way. In many important applications, striding memory behavior is an important component and we believe we are the first to connect the address trace analysis problem with the Hough transform. The resulting analysis is surprisingly robust to noise and can capture general striding behavior. While this approach is effective for the memory problems we examined, there is no shortage of opportunity to build on the techniques we lay out to create more robust and higher quality trace wringing systems. Fully leveraging the best synthetic trace, trace compression, and statistical modeling techniques and understanding what they each bring to the problem is one next step. Bringing the full algorithmic power provided by the fact that *any* public trace data can be leveraged in the compression is also very promising. This opportunity is particular interesting as it sits outside of any past lossy compression or synthetic trace scheme's ability to exploit (i.e. minimizing total data transferred is different than minimizing sensitive data transferred). Further forward, we see a set of access behaviors (uniform random, stride, etc) that might form a set of "basis functions" which then are composed to describe a set of traces. Finding the *best* set of basis functions and how to *optimally* compose them to form good proxy traces can lead to many interesting follow-on works. It remains to be seen just how small of a footprint is achievable, but we believe there are orders of magnitude of improvement left to be had. Luckily, because the data to train such a wringing approach is generated completely by machine, this is an area where there is a great opportunity to gather a great deal of data to inform our models. The exploration of the hyper-parameter space of the wringing process can be automated using existing frameworks (e.g., [70]). In the end, this work is a stepping stone to more general methods for trace sharing and we hope the clear metrics for success (e.g. share as few bits as possible) prompts further discussion and effort by the community.

Chapter 3

Context Matters: Optimizing Privacy of Traces with Information Flow Tracking

As we have seen from Chapter 2, application tuning requires a coordinated effort across hardware and software to achieve optimized application performance. Execution traces offer unique insights into a program's behavior over real inputs and serve as an invaluable resource for hardware and software engineers during the co-optimization process. Unfortunately, these traces are rarely shared between technology partners because even the simplest address traces gathered from applications that utilize private data can divulge sensitive information. The fundamental tradeoff between sharing accurate and precise execution information that will lead to the best co-optimization and protecting sensitive data remains the just as pivotal.

Concurrently, global policy is moving in favor of providing users with privacy protections. As a field, we must develop tools, mechanisms, and primitives to uphold these regulatory protections. In this chapter, we focus on refining our methodology from Chapter 2 to provide an order of magnitude of improvement in the utility of wrung traces. We (1) utilize the leading industry standard: the LINDDUN privacy threat modeling method, (2) leverage advances in information flow tracking techniques to prevent inadvertent leakage of information, (3) introduce multiple classes of privacy-enhancing tracing techniques that allow context-aware differentiation of what information should remain in the trace and in what amounts based on annotations of private user input, and finally, (4) run non-convex optimization to maximize utility and minimize bit leakage. To explore how meaningful the privatized traces are, we compare cache simulation and prefetching properties. This new approach leaks as few as **zero bits** of sensitive information and has an **order of magnitude better utility** than prior work.

3.1 Introduction

Traces provide invaluable insights into a program's behavior, notably its interactions with the runtime system and underlying hardware. These insights guide system designers as they perform hardware-software co-optimization and application tuning. For example, address traces can be used to optimize memory and cache behavior. Unfortunately program traces also leak practically unbounded and arbitrary information about the programs, sometimes in unexpected ways. This is especially true when program behaviors are influenced by some private user inputs to that program.

For example, consider a trace taken from a machine as it runs a bioinformatics application that analyzes nucleotide sequences for alignment; `hmmersearch` [71] detects the similarity, or *homology*, between a private user profile (containing a nucleotide sequence) and a database of known sequences. As the program runs, the program path and data accesses will be dependent on the user's private nucleotide and that dependence will show up in the trace as certain code paths being exercised as a function of the nucleotide

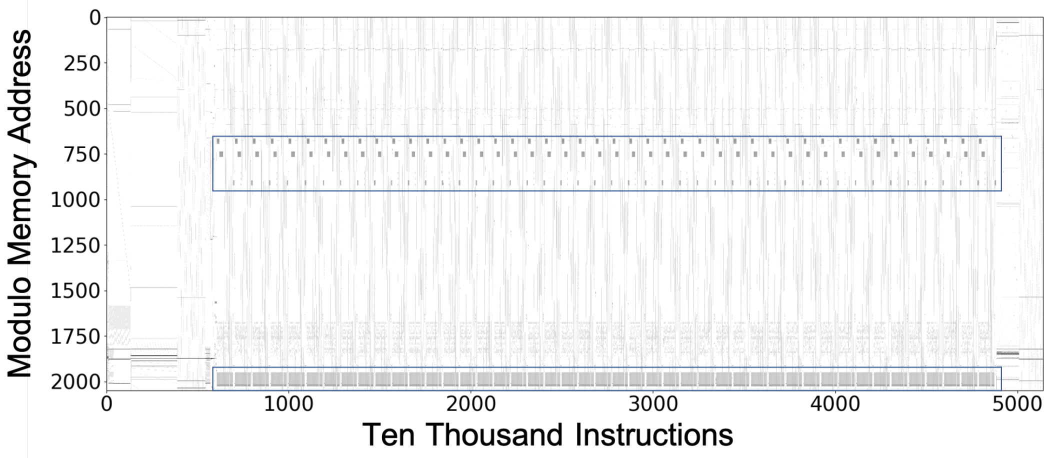


Figure 3.1: Modulo memory heatmaps of `hmmersearch` can reveal number of nucleotide matches between a private user profile and a known database of sequences.

matches. Therefore, by observing the trace, we can make inferences and gain information about the nucleotide matches. Figure 3.1 is a visual representation, or a *modulo memory heatmap* [7] of a `hmmersearch` execution. These heatmaps concisely represent the memory activity of a program execution. In this case, the private nucleotide has 45 matches with the known database. There are also 45 repeating features observable in the highlighted regions. The knowledge of this relation can be used to infer the number of matches as is shown in the highlighted regions in this heatmap!

History has taught us that researchers and engineers are highly effective at inferring sensitive information from even incredibly noisy data. The most advanced methods are capable of combining abstruse statistical artifacts with prior understanding of computations to identify secret keys with surprising effectiveness [72, 73, 74, 75, 76]. Privacy concerns from leaked data extend far beyond cryptographic techniques and include reverse engineering attacks on private neural network parameters [77, 78], web traffic behavior [79], to even the complete reconstruction of images from a few features [80, 15]. While providing a trace can be an effective means of sharing program behavior with others, the same precise visibility into program behavior that makes a trace useful for performance analysis also makes it useful for identifying private and sensitive data.

As a result, while program traces are a very effective means of gaining insights, they remain problematic to share widely as things stand today. Moreover, the privacy of program traces is not well defined and is difficult to formalize. What exactly constitutes leakage in a trace? Is it the individual addresses, the relationship between the addresses, or more? This lack of understanding adds a significant research challenge to this area, making it increasingly unmanageable to share traces even if performance improvements are highly promising.

Prior work [15, 7, 81, 82] has established that for many applications (including sharing of address traces [7]), there exists a tradeoff between sharing data for improvements and profit, i.e. *utility*, and withholding sensitive data, i.e. *privacy*. Studying the privacy-utility tradeoff allows us to move away from operating over an opaque system with no utility and a transparent system where all data is in the public domain. As a field, we must tackle system design with privacy and utility in mind. It is therefore necessary to clearly define goals, threats, and build some amount of tolerance into our systems.

In this chapter, we apply the leading privacy threat modelling framework, LINDDUN [14], to the problem of defining privacy of program traces. The LINDDUN methodology is an information flow oriented model that helps elicit and mitigate threats to the privacy of the defined system. Specifically, we employ the "hard privacy" threat model where the objective is to minimize information leakage. Informed by the LINDDUN framework, we present strategies for better managing the leakage of information introduced by the sharing of program address traces. While prior work has shown how address trace data can be represented with statistical models [46, 22, 38], proxy applications [21], or via trace wringing [7], these trace sharing approaches all treat every part of the trace as *homogeneously* sensitive. We introduce hybrid techniques that find and eliminate or differentially reduces sensitive information from program traces before they are shared, thereby eliminating the flow of sensitive information from the private entities to untrusted processes.

Our hybrid technique, *trace scrubbing*, begins with application-level input sensitivity information and, leveraging techniques from information flow analysis, automatically infers the degree of sensitivity of each memory reference in an execution trace so that its leakage can be managed appropriately. That sensitive data might be location information, genetic sequences, lists of contacts, people in images, or private messages. In any of these domains the specifics of the data considered private, the *degree* of sensitivity, and how it interacts with program execution will inform the decision of what information would need to be reduced or even removed entirely from a trace.

Once a decision has been made about what information to limit, we must consider how this reduction in information will impact the *utility* of the trace. The definition of utility depends on the use case. For our application of generating privacy-enhanced address traces, this is the distance between cache missrates and prefetching in simulations. Does the complete removal of these sensitive references make the program traces less useful? The answer to that question is dependent on both the amount of sensitive data and how much the program behavior depends on it. In certain cases simply removing all potentially sensitive data is possible, but in other cases stand-in information will need to be used to “fill in” for any data removed. We explore a spectrum of methods in this work that allows us to balance sensitive information leakage and utility of the program traces. Specifically our contributions are as follows.

- We formalize the privacy threats to sharing program traces using the LINDDUN privacy threat modeling framework. While this leading industry standard is prevalent in modeling software architectures, this is the first time it is being applied in the context of sharing program traces, and the first time it is appearing in a hardware-software system.

- We show that the problem of address trace privacy can be linked to the well-studied problem of information flow analysis and that by learning which addresses in a trace are sensitive to the private input, we can specifically target them with more aggressive leakage mitigation strategies.
- We demonstrate the resulting technique, which balances the cost of replacements with the set of application behaviors that need to be shared automatically, provides a new and tighter upper bound on leakage by combining non-sensitive and leakage-reduced information effectively. Importantly, we demonstrate that this balance is not tied to the specific *use* of the trace (through a study of prefetching from these traces gathered for cache analysis).

We begin with a discussion of our problem formulation and background on privacy models (Section 3.2). We follow with an overview of our new trace scrubbing technique and outline a specific instance of this idea that builds on multi-execution information flow analysis (Section 3.3). We evaluate our technique on a set of applications (Section 3.4). We end by relating this approach to some additional prior efforts (Section 3.5).

3.2 Modeling Threats to Privacy of Traces

Threat modeling is widely used to identify risk and potential security vulnerabilities and prioritize mitigations. A privacy threat model specifically targets the privacy of the system, rather than the security of the system. In this section, we detail how we utilize privacy threat modeling techniques, and a recent privacy model with high effectiveness when dealing with memory address traces.

3.2.1 Trace Privacy with LINDDUN

LINDDUN [14] is the leading privacy threat modeling framework. LINDDUN is a mnemonic for the privacy threat categories it supports: (a) Linkability, (b) Identifiability, (c) Non-repudiation, (d) Detectability, (e) Disclosure of Information, (f) Unawareness, and (g) Non-compliance. The LINDDUN analysis occurs in 3 steps: (1) model the system, (2) elicit threats, and (3) manage threats. LINDDUN uses data flow diagrams (DFD) to model the system. It elicits threats by iterating over the DFD and manages threats by targeting the riskiest DFD elements. The elicitation and mitigation steps are strengthened by privacy knowledge support structured in to the 7 privacy threat categories encapsulated within LINDDUN's acronym.

We provide a LINDDUN data flow diagram for our system in Figure 3.2. The legend describes the various types of elements in the DFD. In our case, the "User", "Application Designer", "Application", "Trace Generation" process, and the "Trace Database" are trusted entities and lie within the "Trust Boundary" in the figure. The Hardware/System Designer and the Application Tuning" process are the untrusted elements.

The second step is to elicit threats to privacy by mapping the *untrusted* DFD elements onto the threats in its knowledge base. We study the relationships between these untrusted DFD elements to the sensitive DFD elements that we must protect. For this, we ask the following questions; (1) can an adversary link two items of interest without knowing the identity of the data subject(s) involved (Linkability), (2) can an adversary identify a data subject from a set of data subjects through an item of interest (Identifiability), (3) can the data subject deny a claim (e.g., having performed an action, or sent a request) (Non-repudiation), (4) can an adversary distinguish whether an item of interest about a data subject exists or not (Detectability), (5) can an adversary learn the content of an item of interest about a data subject (Disclosure of information), (6) can the data subject be unaware of the collection, processing, storage, or sharing activities of the data subject's personal data (Unawareness), and (7) is the processing, storage, or handling of personal data compliant with legislation, regulation, and/or policy (Non-compliance). (Note: in this work, we do not consider non-compliance.) In Table 3.1, we provide a mapping for our system, which looks at how the untrusted entities and processes interact and what they can learn about the system by looking at a sensitive program trace that might be shared for application-tuning.

Finally, we manage the threats to privacy using privacy-enhancing technologies and LINDDUN provides an extensive taxonomy in its knowledge base. The common theme this knowledge base reveals is that for maintaining privacy, it is pivotal to "remove", "hide", "replace", and "generalize" data and data associations. In this work, we apply these mitigation strategies to the problem of sharing traces (Section 3.3).

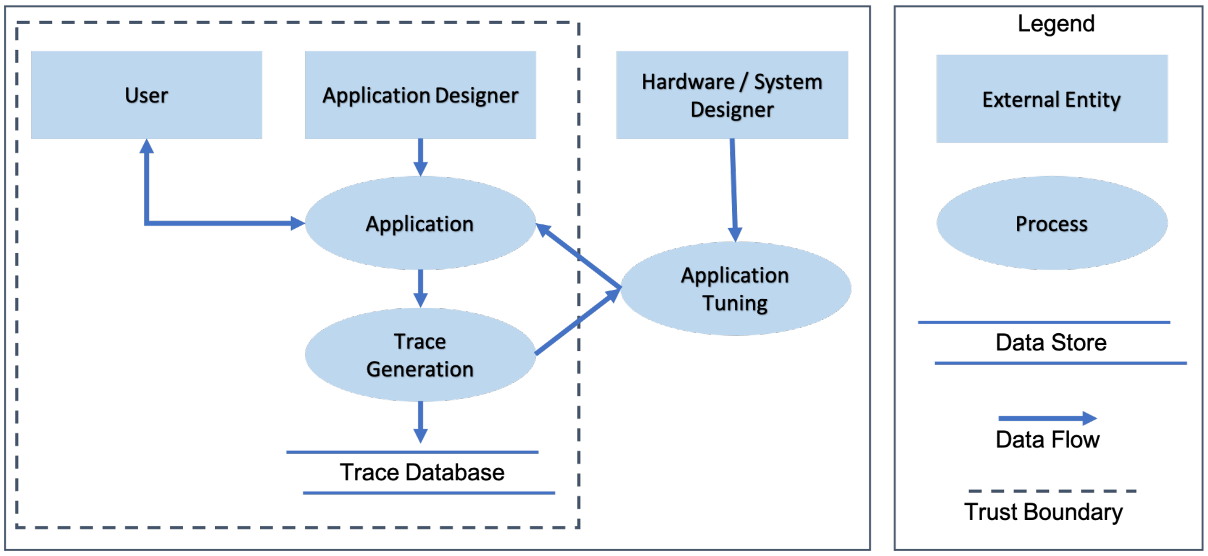


Figure 3.2: Data flow diagram (DFD) for privacy threats to sharing traces for application tuning. External entities, processes, data stores, data flow, and trust boundaries are described through the DFD. Elicitation and mitigation steps iterate over elements of this DFD to manage threats.

Table 3.1: Eliciting threats to privacy using the LINDDUN knowledge base. Following from Figure 3.2, we map the untrusted DFD elements to a privacy threat described by LINDDUN.

DFD Element	Description	L	I	N	D	D	U	N
System designer	Untrusted entity	x	x	x		x	x	-
Application tuning	Untrusted process	x	x	x		x	x	-

3.2.2 Trace Wringing Privacy Model

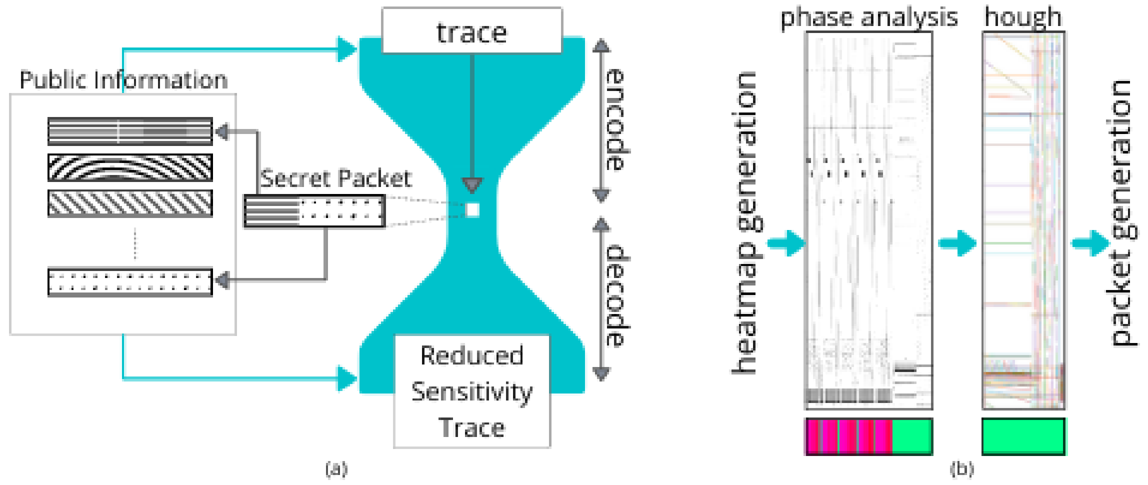


Figure 3.3: (a) In the trace wringing privacy model [7], the secret information is encoded into a compact packet. The size of the packet in bits is the upper-bound on information leaked by the secret trace. Trace wringing argues that if you share n bits about a trace, then you are only leaking n bits. Furthermore, trace wringing allows extensive use of public information. As such, while the information in the packet remains secret, it can be used to point to public information. (b) The trace wringing pipeline [7] is shown here on the `hmmer` benchmark. Wringing begins by generating modulo-memory heatmaps followed by phase analysis using k-means clustering. The cluster labels for each vector is shown in the colorbar under the heatmap. Lines that describe representative phases are then generated using the hough line transform. The phase sequence and line information of representative phases are quantized and compressed into a compact packet.

While superbly useful for purposes of application-tuning, traces can potentially leak arbitrary information about the application, the system, or even users. Power traces and branch traces, which both leak far less precise information than address traces, have been shown to completely compromise critical cryptographic keys [17]. Trace wringing [7] looks at the problem of how to safely share memory access traces and establishes a connection between extreme lossy compression and privacy.

Figure 3.3(a) describes the trace wringing privacy model. The trace with sensitive information is "squeezed" through a narrow n-bit channel into a compact packet. The packet, whose encoding is highly lossy but hopefully captures the most important behaviors of the trace, can then be shared in lieu of the full trace. The size of the packet in bits gives us an upper bound on information leaked by the modified trace (since the modified trace can be completely reconstructed from the packet). One must then attempt to minimize the size of the packet, while still being able to decode the secret packet into meaningfully useful traces. Interestingly the trace wringing privacy model allows, in theory, unlimited use of public information. If there is publicly available knowledge about traces, encodings, programs, or patterns, we may leverage it to minimize bit leakage. Of course any *reference* or "pointer" in to public data derived from an analysis of private data would still need to be included in the packet and would still count against any leakage calculation.

Trace wringing introduced a pipeline of encoding based on computer vision to extract and preserve the underlying behavior of memory access traces. To wring a trace one begins with a modulo-memory heatmap. A modulo-memory heatmap is a 2-D histogram of size N by M which graphically represents the trace behavior with dark spots representing more heavily accessed areas of memory. Here, N is a large power of 2 and M is the number of accesses per ten thousand references. Accesses to the same region of memory over time show up as a horizontal line, while striding behavior appears as a diagonal line. Trace wringing tries to extract and then replay these visible features to reproduce a proxy with the same temporal and spatial locality features. Even complex programs exhibit these patterns and while the striding patterns are important to understand program behavior at a high level, the actual addresses themselves are not. Trace wringing attempts to preserve these“structural” relationships between trace elements and also proportions of phases in which they originally appear. Specifically, the pipeline begins with a phase analysis, recognizing program phases that repeat over time using k-means clustering [62] over the modulo-memory heatmap. The sequence in which phases appear in the program are saved for future replay (in the packet). Detailed striding information is detected and collected by using hough line transforms [64]. The sequence of labels and the detailed hough lines are compressed into a packet which can then be used to generate a proxy trace. An overview of this pipeline is shown in Figure 3.3(b).

Notably, this method treats all addresses in the trace as equally sensitive. Therefore, the number of bits leaked by the trace-wrung proxy are the same for both “safe” and “unsafe” addresses; an AES sbox is treated the exact same as a stack push and the pixels of a face are treated the same as pixels of a patch of sky. There is a pressing need for a full accounting of which addresses are most sensitive so we can definitively, and with finer granularity, reduce the amount of private information while leaving as much low-risk addresses in place as possible. Not only would this yield a tighter upper bound on information leakage, it would also enable higher utility traces since the behavior of non-sensitive addresses could be less perturbed.

3.3 Scrubbing Data from Traces

The privacy threat modeling reveals that to achieve hard privacy, minimizing data and data-association leakage is a key requirement. Informed by LINDDUN and trace wringing (Section 3.2), we actively build in oversight into our system to prevent the leakage using the prescribed “remove”, “replace”, and “generalize” approaches. Specifically, *Trace Scrubbing through Redaction* (Section 3.3.11) embodies the “remove” mitigation principle, *Trace Scrubbing through Replacement* (Section 3.3.12) embodies “replace”, and good “generalizations” can be attained using methods shown in Section 3.4.1.

When minimizing privacy leakage from a trace, there is often no sense of where the most sensitive data is and all addresses in the trace are considered equally “bad to leak”. Accordingly, to bound the overall leakage, the leakage of all the addresses is restricted to a similar degree. However, in real applications, some trace data may be under the influence of extremely sensitive data while others may not. In many cases, only a small subset of the entire set of addresses accessed by an application maybe related to private information. This is one of our key insights: *not all addresses in a program trace are equally bad to leak* and targeting the regions with the strongest information flow from *private and sensitive inputs* is a reliable method to maximize both privacy and utility definitions.

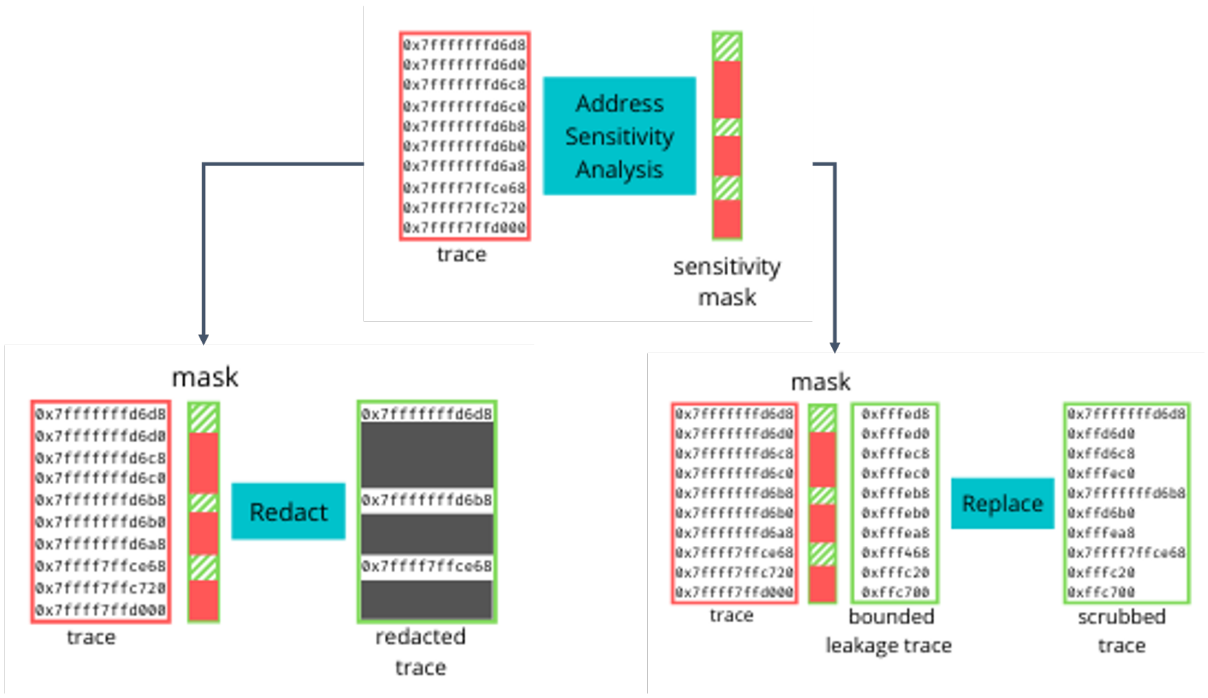


Figure 3.4: Our techniques to scrub sensitive data from traces: redact, and replace. By learning information about locations of sensitive addresses, we can redact them, or replace them with stand-in data with similar behavior but bounded leakage. While trace scrubbing is agnostic to what address sensitivity analysis is used, we use a multi-execution technique. We mark the traces with unbounded leakage in red, and traces with bounded leakage in green.

3.3.1 Trace Scrubbing Strategies

Consider an example of collecting a trace by instrumenting SCP (secure copy) while we copy files over to a remote host. The generated trace will pick up information about the authentication process which uses a private key, `key_0`. An address trace taken during application execution, should never contain the actual key value `key_0`, but it could contain addresses that are influenced by `key_0`. Even in cryptographic algorithms, where the programmers are well aware of side channels introduced by many microarchitecture influences and avoid things like branching based on cryptographic keys, it is still not uncommon for memory addresses to be dependent on key values. Of course concerns about privacy are not limited to cryptographic computations (and our results explore other applications as well), but they provide a very clear example from which we can build intuition.

The idea of information flow analysis is almost as old as the discipline of computer security itself, and here too it finds application. However, one is not concerned about the downstream variables that will be influenced by `key_0`[1], instead we need to understand the influence of private information on load and store instructions including (most importantly) which addresses are accessed. Many information flow analysis techniques start by "tagging" data that is "high" and then observing how it flows through registers and values stored and loaded in memory. Here, by contrast, it is perfectly fine if "high" data is being read from memory; no information is inherently leaked to an address trace in this case. This problem is if the private data is used as part of an address calculation itself, for example, as part of a table lookup or in influencing the order of access to a data structure. While we discuss more of the details of the specific multi-execution information flow analysis used in Section 3.3.2 below and other approaches more in Section 3.5,

[1]unless, of course, those downstream variable influence other loads and stores

the important point to take away is that we can use the outcome of such an analysis to bifurcate the address trace. Some of the addresses will be marked as *dependent* on private data and some will be marked as *independent*. These two classes of address in the trace can then be treated differently. While there are many ways in which one might choose to differentially dial back information from these two different classes of addresses in the trace, we concentrate on two in particular here:

Trace Scrubbing through Redaction

Once we have a trace, and know what the private addresses in the trace are, the simplest and most conservative solution to ensure we are not leaking that private information is to simply delete these addresses from the trace before sharing it. We term this the *redact* trace scrubbing strategy. When dealing with legal documents it is not uncommon to see redactions which often show up as thick black lines covering a portion of the text. When confidential information such as names or places are redacted from classified documents, we can still learn *something* about the redacted words, such as the number of characters by observing the length of the black bars. But, when we redact addresses from our traces, we are not blocking or "zeroing-out" the addresses that are sensitive; we simply do not share the fact that there was any address which leaked information at all. The resulting scrubbed trace is shorter, but if the information flow analysis is sound and all influence is actually captured, then there is no way to figure out where the missing addresses would go from the data released. Redacting these sensitive addresses from the trace leads to an information leakage of zero bits. Note that this will be true for any sound analysis, even if it is not precise.

Of course, redacting hundreds of thousands of addresses from a trace will come at a high cost to utility. By choosing not to share information about sensitive addresses altogether, we lose out on key locality information which may adversely affect the utility of these traces when sharing program behavior. The resulting scrubbed trace is also shorter than the input trace since the redacted addresses are not replaced with any other information. A sound but imprecise analysis might even further exacerbate this issue. Still, as we will see, redacting a trace often has impressive utility as well.

Trace Scrubbing through Replacement

While redaction has the advantage of leaking no sensitive information about the trace, when program sharing requires a higher utility, we need a better strategy. One solution is to replace the sensitive addresses in the trace with similar stand-in data with bounded leakage, instead of completely eliminating it as in redact. A key insight is that we can use our trace wringing privacy model more cleverly, i.e., we can now treat an entire redacted trace as public information and can share it freely. In fact, we can leverage this new public information and build new trace scrubbing strategies on top of this simple redaction method. However, any additional information we provide about the redacted information, *including where in the trace the addresses were redacted from*, will leak additional private information. Replacement tries to get the best of both worlds, it starts with a redacted trace but additional information is substituted into the trace strategically from a trace-wrung proxy.

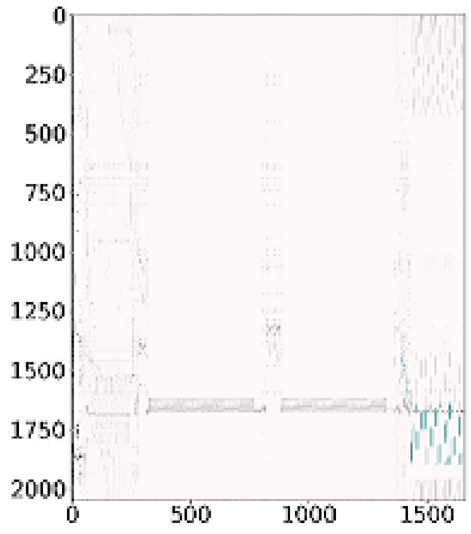
To really maximize the use of the replacement trace scrubbing strategy and find many different points in the bit-error space we can employ replacement incrementally. By starting from a fully redacted trace, and incrementally increasing the number of addresses being included, we can sweep the tradeoff space. One way to do this is to add information back into the redacted trace phase-by-phase. Since the trace wringing pipeline separates information in terms of program phases, we can leverage this and simply add information one phase at a time and generate a superset of the combinations of phases to generate a spectrum of points. This can be especially useful when making decisions about optimality under a tight bit budget. Unlike in redaction, in replace, we also leak information about where the sensitive addresses are located. The information leakage includes the program phases being called as stand-in (the representative phase from trace wringing), the sequence in which they appear across the trace (the phase sequence from trace wringing), and locations or indices where the replacement must happen. We describe our trace scrubbing methods in Figure 3.4.

3.3.2 Inferring Private Addresses

As we have mentioned before, private inputs to an application can influence what a trace looks like. Addresses that hold private information can be tracked using information tracking methods and learn where the sensitive addresses are. An important concept in information flow is that of non-interference [83], where changes to high security inputs affect low security outputs. Researchers have observed that this idea can be extended to measure information flow by counting the number of outputs generated by a program given a set of inputs [84, 85, 86]. Each trace is captured from an actual execution of the program, and we can create a set of such executions that represent different inputs.

Let's consider our running example of collecting a trace by instrumenting SCP while we copy files to a remote server. The authentication in this run happens via the default keys. Now consider that you change your identity file and use `key_1` for authentication, and collect another trace. Can this change of private keys be observed in the traces? Not unexpectedly, yes, the location of the keys is now determined by the identity file and this will show up as different addresses accessed in memory compared to the default keys. Comparing multiple executions can pick up these differences.

While there are many ways to perform information flow analysis, this particular method requires deterministic behavior to be precise. Note that any non-determinism in



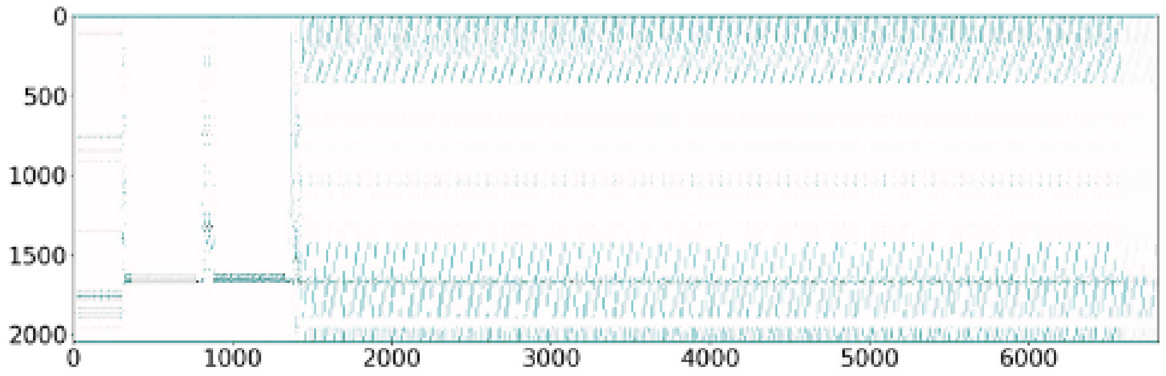


Figure 3.5: SCP heatmaps with an overlay of the sensitive addresses in the trace. The x-axis is windows of size 10,000 instructions and indicates the passage of time; the y-axis is the address modulo 2048. In the first figure, we find the addresses that are sensitive to a change in the SCP keys, and mark them in blue. In the second figure, we find and mark the addresses that are sensitive to a change in data that is being transferred to the remote machine.

the system will simply show up as additional differences and will result in a conservative "over redaction" of the trace. However, non-determinism should still be reduced to maximize utility of the technique under any given leakage restriction. We first collect a trace of the application running with the secret input via Intel's Pin tool [87]. We then re-run this application with a modified private user input to observe how this change influences the generated trace. In order to maintain repeatability across traces, address space layout randomization (ASLR) is turned off during collection and changes to the length of the command line arguments are minimized. This allows us to observe the different program paths taken with changing sensitive inputs to the application. While not used in this chapter, there is significant existing literature on deterministic record and replay methods and techniques [88].

An important side effect of measuring information flow in this way, where the actual system is executed repeatedly, is that side channels in the address traces are captured in the information flow measurements because the definition of the system is the system itself. The only way to hide information flow is if the applications under evaluation are actively communicating across executions such as through the timing or storage objects that are part of the record used for deterministic record and replay [84]. Thus, although measuring information flow in adversarial environments is a notoriously challenging problem, for an application under controlled evaluation where we can apply more advanced information flow measurement techniques, many of the traditional concerns about covert channels and system definitions are much less significant because we can apply techniques that captures most side channels.

Once the traces are gathered, there is the problem of understanding their differences. Because the traces can be of different lengths, the addresses may not be strictly subsets of one another, and there is no clearly defined shared reference points, any simple direct address-by-address comparison is not possible. Ideally we would examine the minimum cost way that one trace might be "edited" into the other to identify the differences, but because these traces are billions of instructions long that approach is computationally prohibitive. Instead we examine differences conservatively using standard file comparison tools [89] between stripped down traces generated from multiple executions with changing private inputs. Again this form of comparison of differences is sound but not precise as there could potentially be smaller "explanations" of differences between the traces.

With this difference map in hand, going back to our example of SCP, we can visualize the impact of changing a sensitive input. In Figure 3.5, we show where these multi-execution differences appear and overlay them on top of the existing heatmaps for SCP to understand what is changing over multiple executions. In the first figure, we observe what memory accesses are sensitive to the private keys being changed from the default keys to `key_1`. The percentage of sensitive addresses here is about 2.5%. Next, we change what we consider to be the private input to our analysis. We now consider the data being transferred via SCP to be the sensitive input. The trace region that is sensitive to change in data being transferred is much higher: 90%.

3.4 Evaluation

3.4.1 Collecting Bounded Leakage Traces

We compare the privacy-utility performance of trace scrubbing methods to prior work on trace wringing [7]. We begin by demonstrating that prior work on trace wringing actually describes a previously unexplored Pareto surface in the dimensions of privacy and the utility of memory access traces. We, for the first time, formalize the approach as an optimization problem and quantitatively evaluate this tradeoff space to show that order of magnitude improvements over prior work are possible. We briefly describe our methods here.

We seek a solution to the optimization problem min $J(\bar{h})$, and J can be appropriately defined, e.g.:

$$J_1(\bar{h}) = \text{error} \tag{3.1}$$

$$J_2(\bar{h}) = \text{bits} \times \text{error} \tag{3.2}$$

where $\bar{h}$ represents the tuple of parameters we wish to optimize, bits refers to the size of packet we are sharing, and error is measured as deviation from cache missrates of the ground truth. Specifically, the six parameters we optimize are:

- Number of clusters, k, in k-means clustering, to determine the number of program phases there might be in the heatmap of the trace.

- Progressive probabilistic hough transform search parameters *threshold*, *line gap*, and *line length*. These parameters are used to search for striding behavior within program phases. The *threshold* is used to separate lines from surrounding noise, the line length is the minimum length of line segment we are searching for, and the line gap is the maximum allowed distance between points to still be classified as a line.

- Proxy trace generation parameters *filter percent* and *block size*. Filter percent is the percentage of detected hough lines that we ignore when generating addresses, and blocksize is the number of addresses called sequentially.

.

We measure the information leakage as the size of the packet being shared in bits. The packet contains a sequence of program phase labels (seq) and hough lines that describe striding behavior for each representative program phase (rep_phase). This information is quantized, encoded, and compressed. That is,

$$\begin{aligned}\text{I(packet)} =& \text{I(RLE(seq))} \\ &+ \sum_i \text{I(hough\_lines(rep\_phase}_i\text{))}\end{aligned} \tag{3.3}$$

where I is a function measuring the number of bits, RLE is a function performing run-length encoding, and hough_lines is a function returning a set of tuples $\{(x_1,\ y_1,\ x_2,\ y_2,\ w)\}$ indicating the start-end points and weight of the corresponding hough line.

To quantify utility, we perform trace-based cache simulations [67] and compute both the mean absolute error (MAE) and mean relative error (MRE) as:

$$MAE(T_{\text{MR}}, P_{\text{MR}}) = \frac{1}{8}\sum_{i,j} |T_{\text{MR}}(i,j) - P_{\text{MR}}(i,j)| \tag{3.4}$$

$$MRE(T_{\text{MR}}, P_{\text{MR}}) = \frac{1}{8}\sum_{i,j} \frac{|T_{\text{MR}}(i,j) - P_{\text{MR}}(i,j)|}{T_{\text{MR}}(i,j)} \tag{3.5}$$

where $T_{\text{MR}}(i,j)$ and $P_{\text{MR}}(i,j)$ represent the cache missrates for i-way associativite cache ($i \in \{1,\ 4\}$) and cachesize $j \in \{8,\ 16,\ 32,\ 64\}$ (KB) for the ground truth and proxy traces, respectively. For optimization, we use MAE and report MRE (Figure 3.7).

To descend on this objective (Eq. 3.1), we use differential evolution as a search strategy [90]. Differential evolution is a simple, yet powerful non-gradient stochastic algorithm used for global optimization. Starting from an initial seed, the optimizer maintains a population of candidate solutions and during each pass, the algorithm creates new trial candidates by combining existing ones, using the update rule:

$$b' = b_0 + F \times (\text{pop[rand0]} - \text{pop[rand1]}) \tag{3.6}$$

where pop[rand$k$] represents a randomly chosen member of the current population, and the constant mutation factor F or differential weight is used to control the search radius. A large F will increase the search radius, but tends to slow down convergence. The optimizer attempts many hyperparameters settings and finds those that minimize the error (Eq. 3.1), or the bit-error product (Eq. 3.2). Once we have explored the tradeoff space, we can define the surface of useful points, i.e., the ones the leak the least amount of information without excessively compromising the utility.

The plots in Figure 3.7 show the bit-error tradeoff space for our benchmarks. We let the optimizer choose combinations of parameters that minimize the cache missrate error of the generated proxy trace. In the logscale bit-error plot, we mark the Pareto frontier as `wring_opt`. These are the best points in this space. We also compute the centroid of all bit-error points to approximate hand-picking these parameters and mark it as `wring_centroid` in the figure as a reference for what an "un-optimized" parameter selection process might choose. Please note that this figure is log-log. While we present results for a larger set of applications later in our evaluation, it is important to point out that a well chosen point can easily *both* leak a factor of 10x less information *and* a have a factor of 10x less error than a point chosen haphazardly. Using the multi-execution technique, we mark all addresses that are influenced or "tainted" when a change in an annotated private user input occurs. We describe more details of the applications studied below, including the private user input. We collect all our program traces on an Intel(R) Core(TM) i7-7700 CPU with 64GiB system memory running Ubuntu 18.04. Program traces are collected with Intel's Pin tool [87].

3.4.2 Applications

We study seven applications where the user's private data could be exposed. Additionally, we define the annotated secret which is used to mark the sensitive addresses in the trace.

The specific applications we choose are: `scp`, `hmmersearch`, a language model in Python (py), MNIST inference, and video compression using `ffmpeg`. The trace scrubbing strategies track and eliminate the influence of sensitive inputs from appearing in the share-able proxy trace. A brief summary of the applications and input pairs are described in Table 3.2.

SCP (keys)

Secure copy [91] (SCP) is a utility that copies files between local/remote hosts over a secure, encrypted network connection. SCP uses public-key cryptography for user authentication. For password-less authentications, the user's public key must be available on the remote host and the matching private key available locally. We use different authenticating keys to determine what regions of the memory are influenced by the private keys.

SCP (data)

Here, we annotate the data being transferred to be private. We first transfer a secret audio recording in mp3 format. In the second run, we transfer a secret JPEG image. As shown in Table 3.2, the taint from annotating the keys is 2.5% and from annotating the transferred data is 90%. These traces, while collected from the same application, have very different responses to trace scrubbing techniques.

Table 3.2: Benchmarks used to evaluate trace scrubbing.

Benchmark	Private info.	# Addresses	% Sensitive
scp_keys	auth. key	16,613,181	2.5
mnist_weights	trained weights	74,371,421	2.66
mnist_act	activation function	70,338,220	19.54
ffmpeg	video being compressed	75,481,059	38.98
py	training corpus	35,122,862	76.76
scp	data copied	68,177,397	90.23
hmmer	protein seq.	51,392,655	94.15

Training a language model in Python

There are privacy concerns when personal data is used to train large models, for example, in text prediction. One such algorithm is the unsmoothed maximum-likelihood character level language model. We annotate the text corpora [92] as private.

Hidden Markov Sequence Analysis

HMMER [71] is a bioinformatics software for sequence analysis using profile hidden Markov models (HMMs). It is used to identify protein or nucleotide sequences for alignment. HMMER detects homology (similarity) by comparing a profile-HMM to a database of sequences. For this benchmark, we compare a profile-HMM to a database of sequences and annotate the profile-HMM to be private. One of these has zero matches with the database, while the other has 45 matches.

Trained weights

Optimal neural network weights are vital for the accuracy of the network and are also considered trade secrets. We annotate the weights to be private in a network trained to identify handwritten digits (MNIST [93]).

Network architecture

Similar to trained weights, neural network architectures are also considered private and treated as a trade secret. In this benchmark, we annotate the activation function to be private in an MNIST handwritten digits network.

Video compression

Video compression utilities such as `ffmpeg` [94] are content dependent, relying on information shared within and across image frames to achieve compression. Here, we annotate the private video and track changes across multiple compression runs.

3.4.3 Results

We look at the privacy and utility characteristics of the different privacy-preserving trace sharing methods introduced in this chapter: optimized trace wringing, trace scrubbing through redaction, and trace scrubbing through replacement.

Measuring privacy

Our privacy metric is simple but information theoretically sound: we count the total number of bits made available publicly after trace scrubbing. When optimizing trace wringing parameters, the information leaked is the size of the compressed packet which contains information about the program phases, how they occur in time, and the lines that describe the striding behavior within each program phase. Because trace redaction removes all tagged information, and because we assume that all sensitive information is tagged at input, trace redaction leaks no information at all. By eliminating all the sensitive information from the redacted trace, we reduce the trace to a fully publicly disclosable state. If there is still some sensitivity to releasing the remaining data, that

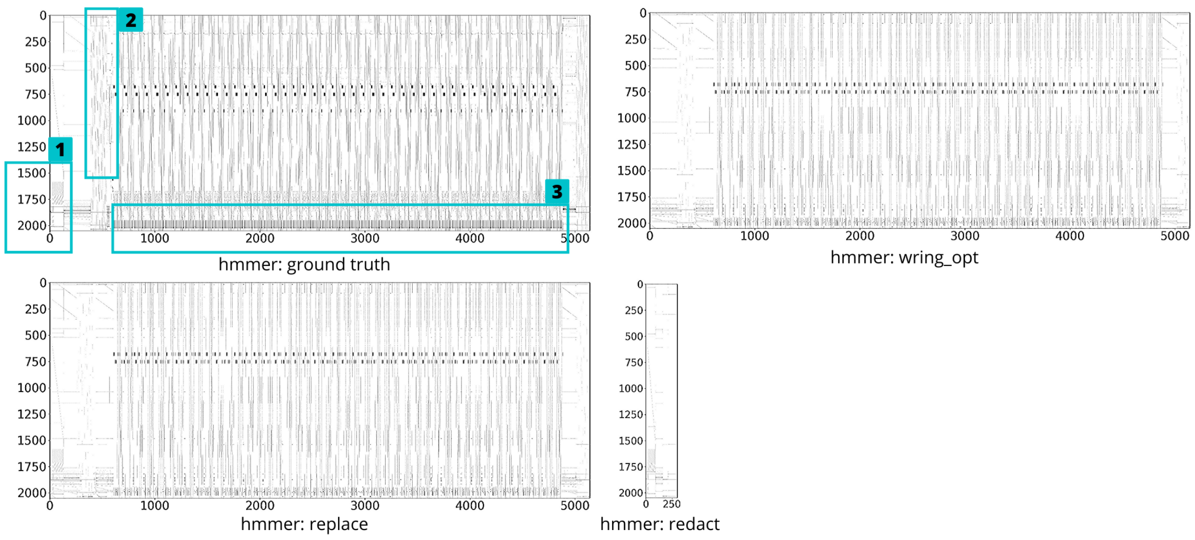


Figure 3.6: Comparing heatmaps generated from trace wringing, and trace scrubbing strategies replace and redact, to the ground truth. The x-axis is windows of size ten thousand instructions and the y-axis is the address modulo 2048. These heatmaps are collected from the `hmmer` benchmark and specific regions of these heatmaps leak information about the program execution visually.

new concern could be further managed through the serial application of trace wringing, but this trivial composition of techniques is not studied further here. The replace strategy, on the other hand, adds information back into the redacted trace replace any data removed with bounded information proxy data. This might be done one program phase at a time or though a combination of different phases. The resulting packet contains information about the phases that occur, where they occur in the redacted trace, and a lossy representation of data within each of the transmitted phases. In either case, replace or redact, the upper bound to information leakage is simple to compute (i.e. count the number of bits in the packet) and thus easy to put into practice and verify in real-world applications. We compare our bit leakage numbers with prior work on Mocktails [95] and present these results in Table 3.3. Our leakage is between 100-$40,000$x smaller.

Measuring utility

All the new techniques presented in this work are evaluated in the context of memory access traces: trace wringing optimization, multi-execution techniques to determine information flow through addresses, and redact and replace trace scrubbing strategies. We use cache simulations and prefetching as utility functions. For a robust and clear understanding of the utility of our traces, we compare cache missrates in eight configurations. We choose four cache sizes, and 2 cache associativities. We measure miss rates for a direct-mapped cache and a four-way set-associative cache; the cache sizes we use are 8KB, 16KB, 32KB, 64KB. For each individual configuration, we measure the absolute and relative errors. Per benchmark, we compute a mean over all configurations to get an absolute and relative error. In Figure 3.7, we present the relative errors as a utility metric.

Qualitative Analysis

While cache miss rates and prefetching are robust quantitative metrics to measure the utility of the bounded information traces, we begin with heatmaps of these traces to get a better qualitative sense of how they are altered by trace scrubbing. We present four modulo-memory heatmaps for `hmmer` in Figure 3.6 for the ground truth trace, the wrung trace with optimal parameters, a replaced trace, and the redacted trace. The x-axis is windows of $10,000$ instructions (time) and the y-axis is the address modulo some large power of 2. Here, the large power of 2 is 2048. These heatmaps show the memory activity over the course of the application's run.

To maintain similar program behavior to the ground truth, the proxy or scrubbed traces should visually resemble the ground truth. Both wring_opt and replace share heatmap similarity, but redact is just barely 5% the length of the ground truth trace.

Most of the “sensitive” activity is also missing from the redacted trace, but these include important regions which could benefit from some optimization through the release of trace data.

Taking a closer look at these traces, we have marked three sections to go over. In section [1] in the ground truth trace, we see some “block-like” features which trace wringing obfuscates. But, since that is not part of the “sensitive” addresses, we see that it shows up in both replaced and redacted traces. Section [2] exists in the ground truth, but is also omitted by trace wringing. Unfortunately though, since this is part of the sensitive information, we do not see it in replace, nor in redact. Most interesting of all is section [3]. The ground truth trace is taken from an application run where there are 45 protein sequence matches against the database. The number of “blocks” seen in section [3] is also forty-five. This means that only by looking at the heatmap, we can learn information about the number of matches found by `hmmersearch`. While this information is captured in the trace-wrung version, the number of “blocks” here is very unclear. Similarly, we see the same behavior in replace. Of course, since this is sensitive information, it does not at all show up in the redacted trace.

Quantitative Analysis

Figure 3.7 illustrates our main results. We compare trace wringing, optimized trace wringing, trace redaction, and trace replacement techniques in the bit-error space. The x-axis is the upper bound bit leakage in logscale, and y-axis is the mean relative error in logscale. Each point, represents the bit-error value for a generated proxy trace. The markers represent the various methods used in the work. The wring points and the wring_opt points are the result of the optimization of trace wringing. We mark the centroid of all the wring points to represent naively wringing a trace with hand-picked parameters. Trace redaction is treated as a line, since it leaks zero bits and is considered

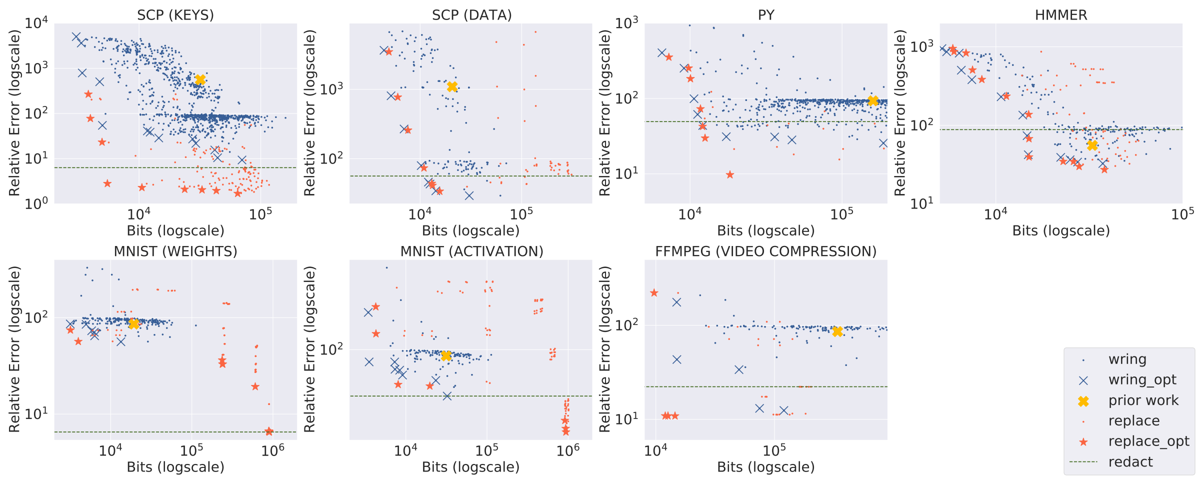


Figure 3.7: Trace scrubbing techniques placed in bits-error space, where the x-axis represents upper-bound information leaked in bits (logscale) and the y-axis represents the mean relative error in cache missrates (logscale). A redacted trace is considered public information and leaks zero bits, therefore, we present it as a line here. Each plot shows the trace wringing points, the replace points, and the redact line. The wringing points include the Pareto frontier, the centroid (naively-handpicked configuration), and all other attempted points. Replace points with information added in phase-by-phase are also presented.

public information according to our privacy model. The replace points are generated by adding information phase-by-phase to the redacted trace. We choose phases from the `wring_opt` points to add back into the redacted trace. We mark the best replace points as `replace_opt`.

For all benchmarks the formulation of trace wringing as an optimization problem and the application of more rigorous techniques takes us quite far. In all cases `wring_opt` *significantly* outperforms `wring_centroid`. The improvement is nearly an order of magnitude better in *both* the x- and y- dimensions compared to `wring_centroid` in most cases. Simply deleting the sensitive addresses in the case of redaction, as expected, leaks no information at all. In terms of miss rate, it performs well in most cases however adding information back into the redacted traces phase-by-phase using the replace trace scrubbing strategy yields better results.

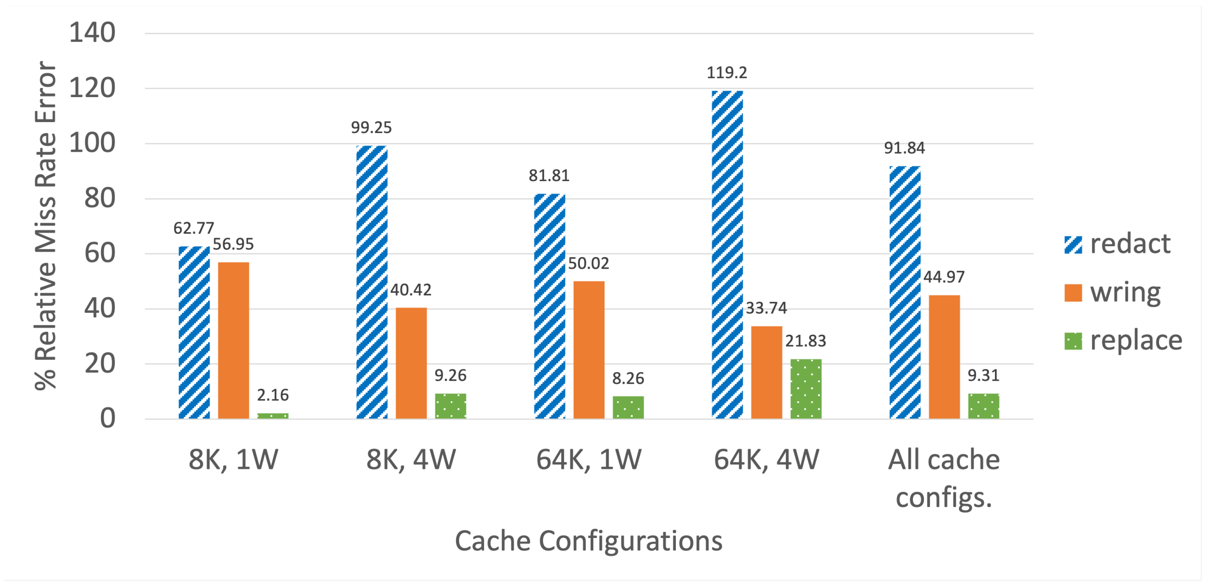


Figure 3.8: Cache miss rates across various cache configurations for `ffmpeg`.

Prefetching on Scrubbed Traces

To demonstrate that the scrubbed traces have utility that extend beyond cache miss rates alone, we evaluate their "prefetchability", i.e., how similar the scrubbed traces are to the ground truth in terms instructions per cycle (IPC). We use Champsim [96] which simulates a 4-wide out-of-order processor with an 8-stage pipeline, a 128-entry reorder buffer, and a 3-level cache hierarchy. We collect IPCs with a next-line L1D prefetcher and the relative errors are posted in Table 3.4. With redaction, IPC errors are between 19-64% of the ground truth; with wring, the errors are within 0.006-0.13%; with replace, the errors are within 0.005-0.12%.

3.5 Related Work

As described above, prior work on trace privacy has treated all trace information as equally "bad to leak". Trace scrubbing advances this idea to specifically target parts

Table 3.3: Comparing bit leakage of trace scrubbing techniques with Mocktails [95]. We measure Mocktails leakage through size of generated profiles. We choose the smallest sizes, and the preferred profiles (2L-TD) for comparison. Note that Mocktails was not designed to minimize bit leakage.

Benchmark	Mocktails [95]		Trace Scrubbing		
	min. leak (KB)	2L-TD (KB)	min. leak (KB)	max. acc (KB)	redact (KB)
scp_keys	3208	8417	13	15	0
scp_data	3143	7852	5	63	0
hmmer	6725	24238	15	37	0
py	37356	122856	12	18	0
mnist_weights	494	1359	3	32	0
mnist_act	487	1350	3	63	0
ffmpeg	257870	564967	12	14	0

of the traces that are sensitive to private information flows and in doing so builds on concepts and techniques in closely related fields of information flow tracking, synthetic trace generation, and program privacy techniques which we describe in more detail.

Information Flow Analysis. Information flow analysis via software-only techniques [97, 98, 99, 100], programming-languages and analysis based models [101, 102, 103, 104, 100], and through hardware extensions [105, 106, 107, 108, 109, 110, 111] provide the ability to understand the flow of influence in a program. Tags can be used in the enforcement of a policy, such as in the taint analysis [112, 113, 114, 115, 116, 117] where tag propagation methods carry information transitively through a computation to tell us if any specific words are derived from that secret or untrustworthy information. More recent approaches have sought to establish more precise quantitative information flows [58, 116] which could, in theory, even further enhance the ability to make tradeoffs. Our approach is agnostic to the specific information flow analysis used, although questions of soundness

Table 3.4: IPC numbers for the ground truth (GT) and relative error percentage of redacted traces (rd), replaced traces (rp) and wrung traces (wr). The settings are chosen to maximize accuracy (.a) and minimize bit leakage (.b). Note that we did not rerun optimization to find these parameters, and chose them from the previous study on cache missrates.

Exp.	IPC	% error in IPC				
	GT	rd.	rp.a	rp.b	wr.a	wr.b
scp_keys	1.053	19.61	0.005	0.006	0.031	0.028
scp	0.939	38.90	0.014	0.025	0.020	0.022
py	0.696	63.89	0.017	0.024	0.020	0.006
hmmer	1.244	18.94	0.121	0.009	0.118	0.133
mnist_weights	1.037	9.67	0.038	0.070	0.048	0.045
mnist_act	0.971	20.59	0.002	0.001	0.001	0.001
ffmpeg	0.964	54.67	0.804	0.737	0.788	0.406

and precision will certainly have downstream ramifications that could require even deeper tradeoff analysis. The information flow techniques applied in the work are based on secure multi-execution methods [85] which introduce a *provably sound and precise method* for ensuring that a program is noninterferant (outputs are not influenced by inputs at higher security level). Related efforts [84, 86] developed an information flow security technique that can detect information theft and provide an upper bound on amount of information that can be stolen without being detected.

Synthetic Trace Generation Characterizing and benchmarking performance[95, 38, 46, 22, 118, 119], when workloads do not yet exist or when systems are too complex to simulate, is done via synthetic trace generation and statistical simulation techniques. For early performance models of big-data applications, CAMP [120] proposes a system-level proxy benchmark generation methodology to accurately both model core and memory. WEST [119] utilizes reuse distance to model temporal and spatial locality by building multiple models per cache configuration (cacheline size, etc.). SLAB [118] leverages instruction streams to improve accuracy and reduce metadata size. CAMP proxies are approximately 10-12x shorter than the originals and have an average cloning error of 11%. SLAB metadata is reported to be 7% the size of LLC traces and have 91% accuracy. WEST produces profile sizes for individual cache sizes and for a 32KB L1, 8M L2 cache, the profile size is 233KB. MeToo [121] extends WEST by simulating DRAMs and produces 20-50x shorter traces with an average error of 4.2%. While all of these schemes were intended for easier sharing and more compressed representations of traces and not privacy, it is conceivable they could be further tuned to bring the amount of data shared down to the order of a few KB we share with our best tuned approach treating all addresses as equally sensitive. However, as better techniques for representing traces are developed, the *additional* use of redaction and other techniques differentiating the sensitivity of particular addresses relative to one another (as we describe in this chapter) should strictly improve the accuracy/privacy tradeoff achievable.

Enhanced Program Privacy We should also point out that the performance trace analysis of an entire execution is somewhat related to, but quite distinct from, the problem of sharing information for bug reports. For example, it is possible to generate bug reports that include new inputs that made the software fail in the way it originally did [56]. In that work it is further shown how to compute an upper bound on the information leaked by the bug report and present this to the users to assist them in making the decision to share the report or not. RESPA [122] is another example that generates anonymous error reports using symbolic execution to find failure-inducing paths and derive the conditions to replay the execution. With the goal of automatically rectifying dangerous inputs, SOAP [123] first learns a set of constraints characterizing typical inputs and when given an atypical input that does not satisfy these constraints, which SOAP then automatically rectifies. Scrash [124] modifies the source of C programs and safeguards to enhance user privacy by removing sensitive information from application crash reports. Other prior techniques have described how to transform program execution traces to maximize users' anonymity, using a crowd of users and k-anonymity [125] to eliminate personally identifiable information or protect the privacy of cloud-hosted deep neural network inference [126].

Chapter 4

Wringing Beyond Traces: Mitigating Reverse Engineering Attacks in Computer Vision Pipelines

In this chapter, I demonstrate how wringing generalizes beyond program traces. Specifically, we look at emerging technologies such as autonomous driving and augmented reality (AR), where the privacy of visual data is a critical concern. For example, these applications rely on localization based on user images derived from always-on cameras and sensors. Localization is the process of identifying where in three-dimensional space the object of interest is. It reveals the location and pose of objects of interest such as an autonomous car or a person wearing AR glasses. The widely adopted technology uses local feature descriptors at specific key points (e.g. corners) which are derived from the images. It was long thought that once extracted, feature descriptors could not be transformed back into the images they were derived from. However, recent work [80] has demonstrated that under certain conditions reverse engineering attacks are possible and allow an adversary to reconstruct RGB images. **This poses a risk to user privacy**. In this chapter, we model potential adversaries using a privacy threat model and we believe that we are the first to have performed such threat modeling in the computer vision field. Subsequently, we show under controlled conditions a reverse engineering attack on sparse feature maps and analyze the vulnerability of popular descriptors including FREAK, SIFT and SOSNet. Our reconstruction is state-of-the-art and produces high quality reverse-engineered images; it underscores the importance of prioritizing privacy-preserving mechanisms for localization. Finally, we evaluate potential mitigation techniques that employ the data minimization philosophy to attain privacy: we minimize the information leaked by selecting only a subset of descriptors and carefully balance privacy reconstruction risk while preserving image matching accuracy, a vital step in localization. Our results show that similar matching accuracy can be obtained when revealing less information thereby demonstrating that wringing generalizes beyond traces, and that privacy-preserving localization is possible.

4.1 Introduction

Privacy and security of user data has quickly become an important concern and design consideration when engineering computer vision applications such as autonomous driving and augmented reality systems. In order to support machine perception stacks, these systems require always-on information capture. Most of these use-cases rely directly or indirectly on the data that originates from the user's device, i.e., RGB, inertial, depth, and other sensor values. These data assets are potentially rich in private information, but due to the compute power limitations on the device, they must be sent to a service provider to enable services such as localization, and virtual content overlay. As a result, there is concern that any data assets shared with a cloud service provider, no matter how well-trusted, can potentially be abused [127]. To enable augmented reality in practice, beyond the application functionality, privacy-preserving techniques are thus an important consideration.

In this work, we focus on localization as a fundamental component of augmented reality. Localization relies on visual data assets to make a prediction of the location and pose of the user; in particular, most established algorithms rely on local feature descriptors. Since these descriptors contain only derived information, they were long thought to be secure.

Unfortunately, recent literature shows that descriptors can be reverse engineered surprisingly well. We show an example in Figure 4.1. In general, a reverse engineering attack is the process by which an artificial object is deconstructed to reveal its designs, architecture, code or to extract knowledge from the object [128]. For feature descriptors, a reverse engineering attack attempts to reconstruct the original RGB image that was used to derive the feature descriptors. The fidelity to which the original RGB image can be reconstructed roughly correlates to the severity of the potential risk to privacy. Prior work [129, 130, 131, 132] has shown that feature descriptors are potentially susceptible to such an attack under a range of conditions and configurations. However, there is limited work on quantitatively analyzing privacy implications as well as evaluating potential defenses against such reverse engineering attacks, which our work will explore.

To scope the problem, we first outline a privacy threat model [133] to contextualize the practicality and data assets available to a descriptor reverse-engineering attack. Using these assets, we show potential reverse engineering attacks and quantify the information leakage to evaluate the privacy implications. We then propose mitigation techniques inspired by some of the current best practices in privacy and security [14]. In particular, we propose two mitigation techniques: (1) reducing the number of features shared and (2) selective suppression of features around potentially sensitive objects. We show that these techniques can mitigate the potency of reverse engineering attacks on feature descriptors

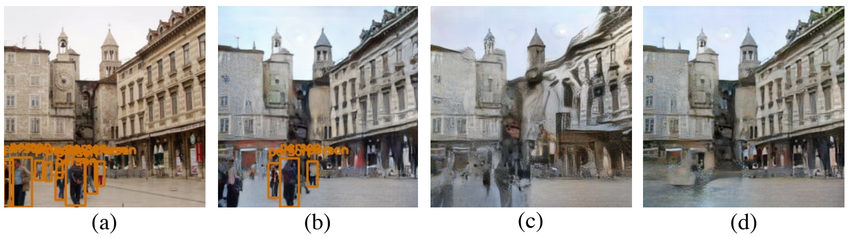


Figure 4.1: **Reverse Engineering Attack and Mitigations.** (a) Original image. Objects detected marked in orange (b) Reverse-engineered image using our attack. The reconstruction preserves semantic information. By (c) reducing the number of features or (d) selective suppression around private objects, we reduce the efficacy of the attack and improve privacy.

to improve protections on user data. In summary, we make the following contributions:

1. We present a privacy threat model for a reverse engineering attack to narrow down the privacy-critical information and scope the setup for a practical attack.

2. We demonstrate a reverse engineering attack to reconstruct RGB images from sparse feature descriptors such as FREAK [134], SIFT [135] and SOSNet [136], and quantitatively analyze the privacy implications. In contrast to previous work [132, 131], our approach does not take additional information such as sparse RGB, depth, orientation, or scale as input.

3. We present two mitigation techniques to improve local feature descriptor privacy by reducing the number of keypoints shared for localization. We show that there is a trade-off between enhanced privacy (less fidelity of reconstruction) and the utility (localization accuracy). We also show which keypoints are shared matters for privacy.

4.2 System and Threat Definition

In this section, we first define privacy, utility and their trade-offs in the context of localization. We also describe our privacy threat model, which defines assumptions on adversary behavior and the conditions for a practical reverse engineering attack.

4.2.1 Definitions

Privacy. LINDDUN, a popular methodology in academic discussions, looks at the following privacy properties [133]: linkability, identifiability, non-repudiation, detectability, information disclosure, content unawareness, and policy. LINDDUN claims that whenever users share information, one or more of these privacy properties may be at risk. This leads to the notion that minimizing the amount of shared information improves privacy. However, precisely quantifying the impact on privacy is application-specific and can be implemented as a continuum, modulating the amount of information to be shared as required. In this work, references to privacy risk and/or threat applies specifically to reidentification risk, a direct result of the reverse engineering attack; we describe and evaluate the trade-offs in Section 4.5.2.

Utility. Utility captures the accuracy (or performance) of an application. Applications may have multiple utility functions for a well-rounded understanding of the operation. Utility often presents a trade-off with privacy as performance tends to increase with data size, e.g., ML training. We use feature matching recall as a proxy for localization accuracy (Section 4.5.2).

Privacy-Utility Trade-Off. Applying privacy-preserving techniques can adversely affect utility. Ideally, we want high utility and high privacy, but in practice there is a fundamental trade-off between the amount of information one is willing to share and the utility one receives from sharing it. In this work, the trade-off is between the localization

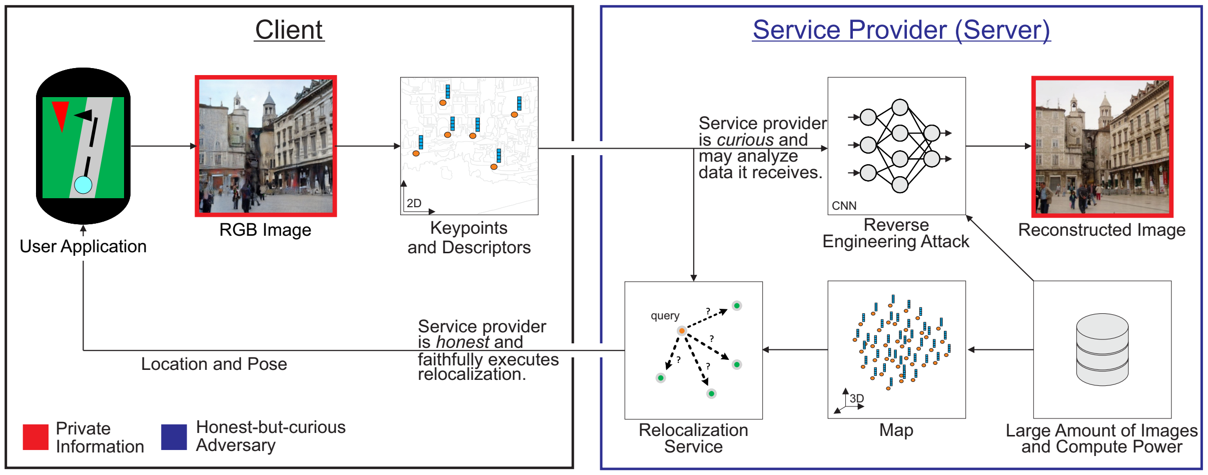


Figure 4.2: **Privacy threat model for localization.** A client derives descriptors from RGB images and shares them with a service provider. The service provider is honest and faithfully executes localization by matching query descriptors against a map. But, the service provider may attempt to derive insight about the user. Our mitigation strategy is to minimize information shared between the client and the service provider to maximize privacy.

accuracy (utility) and the images that may potentially be revealed (privacy), i.e., the features sent to the server are still useful to the application pipeline but do not directly leak the rich information content of RGB images that may contain private user information. In certain cases where the definitions of utility and privacy are simple, this trade-off can be formalized and reasoned about analytically (e.g. k-anonymity [137]). In larger systems this is not possible and we must actively play roles of attacker and defender to model possible attacks and understand the potential risks to user privacy from reidentification. This is the role of a *privacy threat model* [138, 139, 140, 141, 142, 143, 133].

4.2.2 Privacy Threat Model

Building a privacy threat model is application specific. For localization, we use the LINDDUN ”hard privacy” threat model [133] where the objective is to share as little information as possible to an adversary. LINDDUN proposes building a dataflow diagram of a system and marking data assets, adversaries, and potential attack vectors.

These are used to audit against potential threats (described in LINDDUN) that impact privacy. We focus on identifiability, detectability, and information disclosure to audit potential reverse engineering attacks on RGB images. *Identifiability* checks if an adversary can identify items of interest. *Detectability* looks at whether an adversary can detect whether items exist or not. *Information disclosure* asks if private information is disclosed to an adversary without access. An adversary with an RGB image can observe information about each of these properties which poses a risk to privacy. Our goal is to prevent the adversary from having such access.

System Definition and Sensitive Data Assets. Figure 4.2 shows the components of our privacy threat model. Our system follows a client-server architecture to process localization requests. For localization, there are two primary data assets: (1) RGB images (2) feature descriptors. We prevent the sharing of RGB images which can leak private information. Descriptors are perceived as more private and more acceptable to share because they do not *directly* leak RGB information. The client derives feature descriptors (from RGB images) and shares them with the server to query its pose from a global map.

Adversary Definition and Potential Attacks. Our privacy threat model considers the service provider as an adversary (Figure 4.2) that is *honest-but-curious* [144]. This type of adversary is a legitimate participant in the system and executes the agreed upon service faithfully (as opposed to outright malicious behavior). But, while fulfilling the service, the adversary is *curious* and may use available data to learn information about the client. In our case, the adversary might reverse engineer the user's RGB images from feature descriptors. This is possible because the adversary has access to similar data (feature descriptors, source RGB images) and large scale compute resources. The adversary is capable of training deep-learning models (such as a reverse engineering model) to analyze user data in a reasonable amount of time. Our goal is to understand how to improve a client's protection against an honest-but-curious adversary capable of training deep learning models to reverse engineer RGB images from feature descriptors.

4.3 Background

In this section, I define terminology that is widely used in the rest of the chapter but may be unfamiliar to those who are not experts in computer vision.

Localization. In computer vision, localization is the process of locating an object or an instance of an object in an image. This often includes producing a tightly-fit bounding-box centered around the object. In this chapter, we are considering localization in the context of augmented reality which also includes the process of mapping the three-dimensional space the object or person (i.e., agent) is in. Simultaneous localization and mapping (SLAM) [145] is the process of constructing or augmenting a three-dimensional map of the environment around the agent, while simultaneously being aware of where in this map the agent is located.

Relocalization. Camera relocalization, or image-based localization is the process of determining the camera pose, i.e., where and which way the camera is facing in three-dimensional spatial map, derived from the visual scene representation. Relocalization uses a single image to estimate the camera's location and orientation in the three-dimensional space.

Keypoint. A keypoint is a point of interest in an image. A point is chosen to be a keypoint depending on its surroundings, e.g. a corner is considered a keypoint due to the change in intensities around that point. Another defining feature of keypoint locations is that they are scale-invariant, translation-invariant, and rotation-invariant, i.e., even if the image goes through scale, translation or rotation transformations, the same keypoints will be detected.

Descriptor. A descriptor is a finite vector that describes the area of interest around a keypoint. The descriptors capture information about scale and orientation and this information depends on the type of framework used. For example, we use SIFT, FREAK, and SOSNet descriptor types, and each of them capture different information around the keypoints. Some descriptor frameworks, such as SIFT, also have a mechanism to detect their own keypoints, where as frameworks such as FREAK and SOSNet must be supplied with the keypoints.

Feature. A keypoint and descriptor together form a feature.

Feature extraction. In general, feature extraction is the process of dimensionality reduction such that information captured about images (say) are non-repetitive and non-redundant. In this work, feature extraction is the process of finding keypoints and descriptors (features) in an image using one of the descriptor frameworks (SIFT, FREAK, SOSNet).

Image matching.

Scale-invariant feature transform (SIFT). SIFT [135] is a keypoint detector and descriptor. It is a more involved algorithm than finding corners in an image and below I describe the steps. (1) *Scale-space Extrema Detection* which uses Difference of Gaussians (DoG) filtering at different scales to detect blobs of various sizes. This is followed by a search for keypoints, the local extrema, through comparisons to neighboring pixels. (2) *Keypoint localization* This step uses Taylor series expansion of scale space to get more accurate location of extrema, and if the intensity at this extrema is less than some threshold value (0.03 as per the [135]), it is rejected. (3) *Orientation assignment* For rotation-invariance, each keypoint is assigned an orientation using an orientation histogram with 36 bins covering 360 degrees. The highest peak in the histogram is taken and any peak above 80% is also considered to calculate the orientation. (4) *Keypoint descriptor* The keypoint descriptor is created as follows: a 16x16 neighbourhood around the keypoint is taken, divided into 16 sub-blocks of 4x4 size and for each sub-block, an

8 bin orientation histogram is created. A total of 128 bin values are represented as a vector to form the SIFT keypoint descriptor.

Fast retina keypoint (FREAK). FREAK [134] is a keypoint descriptor inspired by the human visual system; especially the retina. A binarized descriptor, it uses a cascade of binary strings which is computed by comparing image intensities over a retinal sampling pattern. FREAK descriptors have the advantage of being fast to compute and have relatively low memory load and are often used for embedded applications.

Second Order Similarity Regularization for Local Descriptor Learning (SOSNet). SOSNet [136] is a learned descriptor with second order similarities (SOS). SOS has previously been used successfully for graph matching, clustering, etc and is known to capture structure and scale information effectively. SOSNet is a convolutional neural network based framework to find descriptors at given keypoints (local descriptors). It utilizes a second order similarity regularization (SOSR) term to the training process that improves matching performance.

4.4 Reverse Engineering Attack

This section defines the convolutional neural network models we use to craft our reverse engineering attack. As shown in Figure 4.2, this model takes sparse local features (keypoints and descriptors) as input and estimates the original RGB image.

4.4.1 Model Architecture

Given a user image $\mathbf{I}(i,j) \in \mathbb{R}^3$ and a derived sparse feature map $\mathbf{F}_{\mathbf{I},M}(i,j) \in \mathbb{R}^C$ containing C-dimensional local descriptors from the image $\mathbf{I}$ using a feature extractor M, we seek to reconstruct an image $\hat{\mathbf{I}}(i,j) \in \mathbb{R}^3$ from $\mathbf{F}_{\mathbf{I},M}$. The sparse feature map is assembled by starting with zero vectors and placing extracted descriptors at keypoint

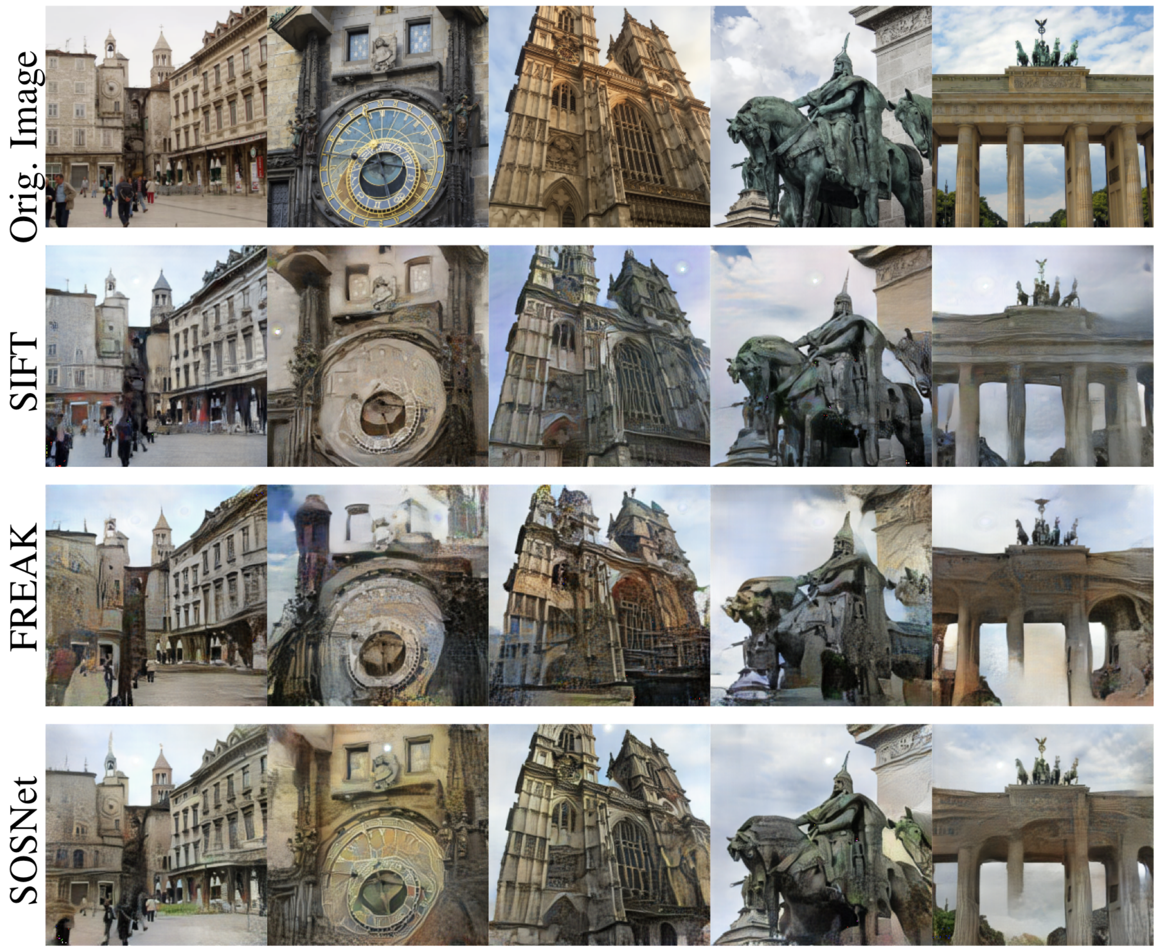


Figure 4.3: **Reverse Engineering Attack Results.** Top to bottom: ground truth and reconstructions from a max. of $1,000$ sparse SIFT, FREAK and SOSNet features. Reconstruction from only sparse local features reveals the original image information extremely well. Note: images show landmarks not included in the training data. Image attribution [146].

locations i, j. Our reverse engineering attack relies on a deep convolutional generator-discriminator architecture that is trained for each specific feature extraction method M. The generator G_M produces the reconstructed image:

$$\hat{\mathbf{I}} = G_M(\mathbf{F}_{\mathbf{I},M})$$

and follows a single 2-dimensional U-Net topology [147] with 5 encoding and 5 decoding layers as well as skip connections with convolutions. The discriminator D_M is a 6 layer convolutional network operating on top of G_M [148]. In order to adhere to our privacy threat model and in contrast to prior work by Pittaluga et al. [132], we do not use depth or RGB inputs and subsequently also do not make use of a VisibNet.

4.4.2 Loss Functions

We use the following loss functions to train the reconstruction network:

MAE. The mean absolute error (MAE) is the pixelwise L1 distance between the reconstructed and ground truth RGB images:

$$L_{mae} = \sum_{i,j} ||\hat{\mathbf{I}}(i,j) - \mathbf{I}(i,j)||_1 \,. \tag{4.1}$$

L2 Perceptual Loss. The L2 perceptual loss is measured as:

$$L_{perc} = \sum_{i,j} \sum_{k=1}^{3} ||\phi_k(\hat{\mathbf{I}}(i,j)) - \phi_k(\mathbf{I}(i,j))||_2^2 \,, \tag{4.2}$$

with ϕ_k being the outputs of a pre-trained and fixed VGG16 ImageNet model [149]. ϕ_k are taken after the ReLU layer k with $k \in \{2, 9, 16\}$.

BCE. For the generator-discriminator combination, we use the binary cross-entropy

(BCE) loss defined as:

$$L_{bce} = \sum_{i,j} log(D_M(\hat{\mathbf{I}}(i,j))) + log(1 - D_M(\mathbf{I}(i,j)))\,. \tag{4.3}$$

Finally, we optimize the losses together:

$$L_G = L_{mae} + \alpha L_{perc} + \beta L_{bce}\,, \tag{4.4}$$

with α and β as scaling factors.

4.4.3 Architecture Implementation Details

Our reverse engineering attack uses a deep convolutional generator-discriminator network (see main paper). We provide the implementation details of our reverse engineering network, including architecture, optimization, and training methodology in this section.

Generator

The generator follows a 2-dimensional U-Net [147] topology with 5 encoding and 5 decoding layers. Specifically, the architecture of the encoder is conv$_{64}$-conv$_{128}$-conv$_{256}$-conv$_{512}$-conv$_{1024}$, where conv$_N$ denotes a convolutional layer with N kernels of size 3×3, stride of 1, and padding of 1. A bias is added to the output, followed by a BatchNorm-2D, and ReLU operation. Between convolutions, there is a 2D MaxPool operation with kernel size and stride both set to 2. The decoder architecture is upconv$_{1024}$-upconv$_{512}$-upconv$_{256}$-upconv$_{128}$-upconv$_{64}$ where upconv$_N$ denotes a convolutional layer with N kernels which is also upsampled by a scale factor of 2. The kernels for these layers are also 3×3 in size and have a stride and padding of both 1. The convolution is also followed by a BatchNorm-2D and ReLU operation.

Discriminator

The discriminator used for adversarial training has the following architecture: Disc_{256}-Disc_{128}-Disc_{64}-Disc_{32}-Disc_{16}-Disc_{8}-Disc_{4} where Disc_N denotes a 2D-convolution with N kernels of size 4×4, stride of 2, and padding of 1, followed by BatchNorm-2D and leaky ReLU with negative slope of 0.2. Disc_{256} is not followed by a batch normalization and in Disc_4 leaky ReLU is replaced by a sigmoid operation.

Training Methodology and Optimization

The loss functions we use are described in Section 4.2 of our paper. Our losses together are described as:

$$L_G = L_{mae} + \alpha L_{perc} + \beta L_{bce} \, , \tag{4.5}$$

where, $\alpha = 1$, and $\beta = 0.1$.

We detail how we use the L2 perceptual loss here. We utilize a VGG16 model pre-trained on ImageNet [150]. The outputs of three ReLU layers are used: layers 2, 9, and 16. ϕ_i is used to denote the these layers. $\phi_1 : \mathbb{R}^{H \times W \times 3} \to \mathbb{R}^{H/2 \times W/2 \times 64}$, $\phi_2 : \mathbb{R}^{H/2 \times W/2 \times 64} \to \mathbb{R}^{H/4 \times W/4 \times 128}$, and $\phi_3 : \mathbb{R}^{H/4 \times W/4 \times 128} \to \mathbb{R}^{H/8 \times W/8 \times 256}$. These outputs are used by the L2 perceptual loss to train the network.

Both the generator and discriminator were trained using the Adam optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and $\epsilon = 1e^{-8}$. The learning rate for the generator is 0.001 and for the discriminator is 0.0001. We train each of the SIFT, FREAK, and SOSNet networks for 400 epochs each. The first 250 epochs are run without the discriminator contributing to the generator-discriminator combination network. The next 150 epochs are run with both the generator and discriminator losses.

4.5 Evaluation

4.5.1 Experimental Setup

Sparse Local Features.

For the feature extraction method M from Section 4.4.1, we use SIFT [135] ($C = 128$), FREAK [134] ($C = 64$), and SOSNet [136] descriptors ($C = 128$) as representatives of traditional and machine-learned variants. Keypoint locations for FREAK and SOSNet were detected using Harris corner detection [151]. For reconstruction, we use the SIFT detector for SIFT descriptors as in [132]; however, for image matching we use Harris corners for SIFT descriptors because we found the SIFT detector performed poorly in this setting.

Training and Evaluation Data. We train our networks on $50,000$ images and their extracted sparse local features from the training partition of the MegaDepth dataset [152]. For testing the reverse engineering attack, we sampled $9,800$ images from the MegaDepth test set that contain objects as candidates for potential private data.

Network Training. A different reverse engineering model M is trained for 400 epochs for each descriptor type. The learning rate is initialized to 0.001 and 0.0001 for the generator and discriminator networks respectively. Learning rates are adjusted using the Adam optimizer [153].

4.5.2 Measuring Privacy and Utility

Measuring Privacy with SSIM. Our first metric for measuring privacy is structural similarity (SSIM), which measures the perceptual similarity between images. In our case, we use SSIM to evaluate how much visual information the reverse engineering attack can recover by comparing against the original image. Therefore, SSIM provides a way to measure identifiability. We note that the SSIM measures to what extent the **whole** image may be recovered, which includes private and public information (e.g. people and buildings respectively); the public information is also available to the service provider when building the map. However, measuring how well the whole image can be reconstructed includes the reconstruction quality of private regions. SSIM can further serve as a proxy to estimate how well other tasks such as object detection, landmark recognition, and optical character recognition may perform on the reverse-engineered image.

Measuring Privacy by Object Detection. We use an object detector (YOLO v3 [16], with 80 classes) to measure how much semantic information can be inferred from the reverse-engineered images. We compare object detection results on both the original and the reconstructed images. If an object's bounding box in the original image has at least 50% overlap with that of the reconstructed image of the same class label, we consider them as a match. The more correspondence between objects in the original and the reconstructed image, the higher the risk to privacy.

Measuring Utility. To assess utility of local features when applying our mitigation strategies, we define an *image matching* task as a proxy for localization and investigate how the feature matching between two images deteriorates as we increase the privacy. Specifically, we generate corresponding image pairs from the 53 landmarks of the test split of the MegaDepth [152] dataset. For each landmark, we sample 50 pairs of images that have at least 20 covisible 3D points determined from a reference map built with COLMAP [154], resulting in $2,650$ image pairs. For each corresponding pair of images, we perform local correspondence matching using input features, and count the number of pairs with at least 20 inlier matches which we deem as successful. We refer to the proportion of image pairs that have been successfully matched as our matching recall, which we use as our utility measure (Table 4.3).

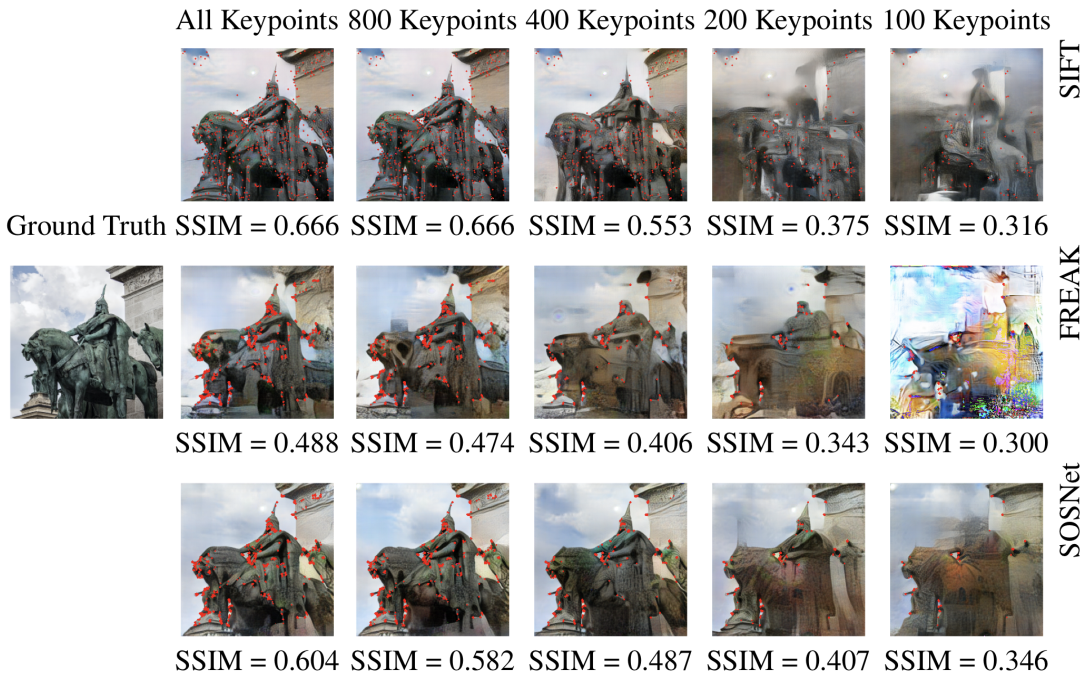


Figure 4.4: **Reverse engineering ablation study of reducing keypoints.** SIFT, FREAK and SOSNet reverse engineering results using $1,000$, 800, 400, 200, and 100 keypoints respectively, annotated in red. Reducing keypoints reduces the potency of the reverse engineering attack. Regions with higher densities of keypoints have better reconstruction quality.

Descriptor	SSIM	Detected Objects
SIFT [135]	0.675	32.58%
FREAK [134]	0.511	19.32%
SOSNet [136]	0.616	41.26%

Table 4.1: **Privacy metrics of reverse-engineered images using** $1,000$ **keypoints.** The number of detected objects using YOLO v3 [16] is measured on the reverse-engineered images relative the number detected on the original images. FREAK descriptors reveal less information than SIFT and SOSNet.

4.5.3 Reverse Engineering Attack

We first evaluate to what extent the reverse-engineering attack from Section 4.4 poses a reidentification risk to privacy. Examples of the reconstructions are shown in Figure 4.3 and the privacy metrics of the reverse-engineered images are given in Table 4.1. Reconstructions using FREAK [134] descriptors yield substantially poorer reconstruction quality and semantic content than SIFT [135] and SOSNet [136]. Despite differences in feature extraction techniques and descriptor sizes, all three descriptors are susceptible to the attack and yield reconstructions comparable to prior work [132] (please see supplemental material for detailed comparison to prior work), but notably without RGB or depth information as input. At a higher level, the results show that under controlled conditions the reverse engineering attack can introduce a reidentification risk of RGB image content. The results from Table 4.1 also show that the reverse-engineered images still allow an adversary to potentially detect and identify some objects that were present in the original images.

4.5.4 Comparison to Prior Work

We compare our work against several prior works that attempt to reverse engineer RGB images from features. Figure 4.5 compares our reverse-engineered image results compared to that of d'Angelo et al. [130] and Weinzaepfel et al. [129]. Compared to the latter, our result using SIFT shown in Figure 4.5 produces a qualitatively better reverse-engineered image with more accurate color estimates. As shown in Figure 4.5, the work from d'Angelo et al. reconstructs image gradients only and is not comparable to our work. We also compare our results to those by Dosovitskiy and Brox [131] in Figure 4.5. In contrast to our work, Dosovitskiy and Brox use more keypoints and descriptors for their reconstruction using SIFT descriptors; they use roughly 3000 keypoints to reconstruct this image while we use $1,000$ or fewer in our experiments. Qualitatively the results are comparable.

The previous state of the art is recent work proposed by Pittaluga et al. [132] which also uses convolutional neural networks to reverse-engineer images. Pittaluga et al. use additional information such as depth and RGB at the keypoint location to supplement SIFT descriptors as input to their reverse engineering model. Our work does not use

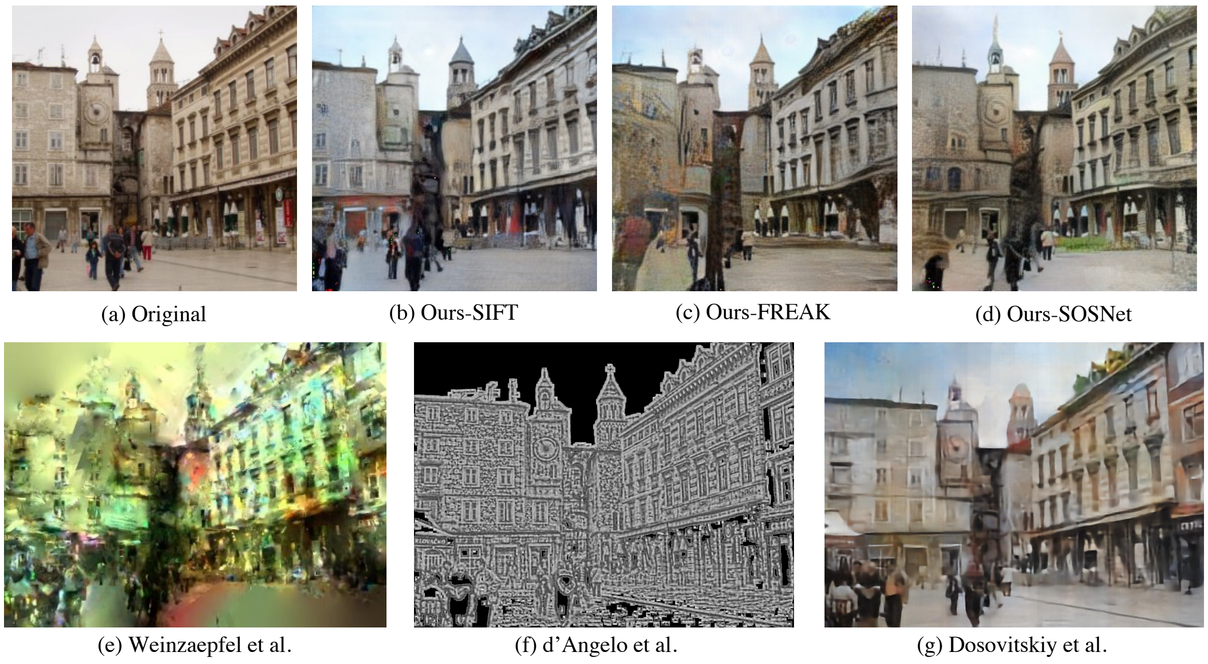


Figure 4.5: **Comparing our reconstruction quality to that of prior work.** Original image (a) and our reconstructions from SIFT (b), FREAK (c), and SOSNet (d) descriptors. Reconstructions by prior work from SIFT descriptors in [129] (e), and BRIEF descriptors in [130] (f), and SIFT descriptors in [131] (g)

	Inputs	SSIM
Prior Work [132]	Depth Only	0.578
	Depth+SIFT	0.597
	Depth+SIFT+RGB	0.631
Ours	SIFT Only	0.675
	FREAK Only	0.511
	SOSNet Only	0.616

Table 4.2: Comparison of average SSIM values of the reverse engineered images from prior work [132] and our work. Our work achieves better SSIM results for SIFT without using inputs like depth or RGB.

depth nor RGB information, and does not make use of a separate network for visibility estimation (as the VisibNet from [132]). We also compare against FREAK and SOSNet descriptors while Pittaluga et al. exclusively analyze SIFT descriptors.

The results show that even without the additional depth and RGB information from Pittaluga et al., our reconstructions produce more detail and more accurate color in average in the cases of SIFT and SOSNet. In contrast, FREAK does not allow us to reconstruct the color information as well and we see some color artifacts (e.g., see the clock image). Since a practical reverse engineering attack for a relocalization service does not provide depth or RGB information to the honest-but-curious adversary, our attack formulation aligns with the real-world scenario. When using all input data assets (depth, SIFT, and RGB) Pittaluga et al. achieve a maximum average SSIM of 0.631 on reconstructions and an average SSIM of 0.578 when using only SIFT descriptors (Table 4.2). In contrast, our reverse engineering attack yields an average SSIM of 0.675 for reconstructions from SIFT features alone and thus provides a new state of the art. We attribute the improvements to our architecture choice and training procedure which we describe below.

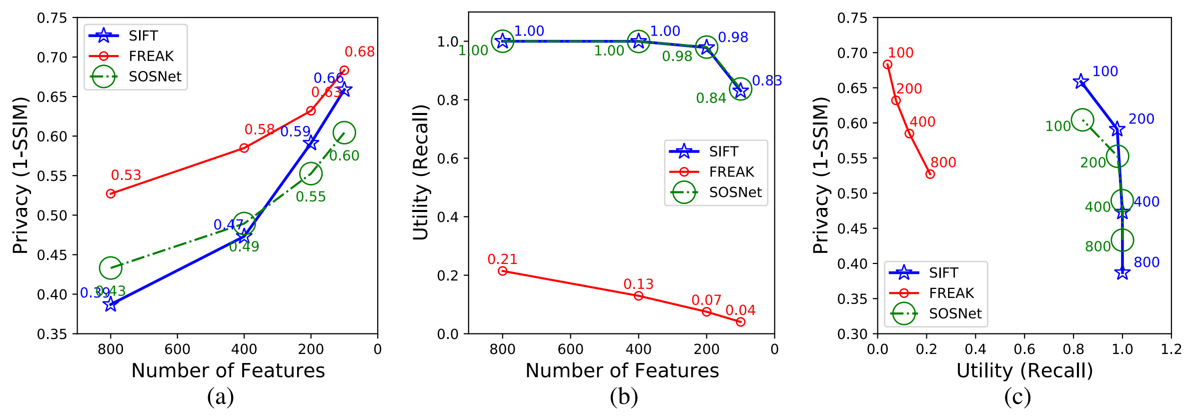


Figure 4.6: **Utility and Privacy Trade-Off when Varying the Number of Features.** Privacy increases when reducing the number of features where FREAK gives the best results. For utility, FREAK and SIFT gives the best results. SIFT gives the best overall trade-off.

4.5.5 Mitigation by Reduction of Features

Following Section 4.2.2, to improve privacy, our objective is to minimize the information shared by the client. To this end, we investigate how reducing the number of features increases privacy at the expense of utility.

For each descriptor type, we retain a maximum of N top-scoring keypoints based on the detector response and vary N from 1000 to 100. For each value of N we then evaluate how well our reverse-engineering models perform. Qualitative results are given in Figure 4.4. We show the average privacy (measured by $1-$SSIM) of the reconstructed images vs. the number of features in Figure 4.6. The data shows the degradation in SSIM of the reconstructed images accelerates as more keypoints are removed. For less than 300 features, SIFT gives better results than SOSNet. FREAK outperforms SIFT and SOSNet, and yields the best results in terms of privacy.

However, despite strong privacy results, FREAK trades-off utility. In Figure 4.6, we show how the utility changes. Here, FREAK gives the lowest utility, indicating that FREAK descriptors overall provide less useful information than SOSNet and SIFT.

Suppression	Privacy (Object Recall)		Utility (Matching Recall)	
	No	Yes	No	Yes
SIFT [135]	20%	**2.21%**	100%	**88%**
FREAK [134]	11%	1.29%	34%	28%
SOSNet [136]	28%	5.21%	100%	88%

Table 4.3: **Privacy-Utility Trade-Off for Selective Feature Suppression.** Object recall shows how many objects can be detected from the reverse engineered images compared to the original images without and with suppression (note that lower is better). Matching recall shows how many images can be successfully matched without and with selective feature suppression. SIFT gives the best overall trade-off.

Interestingly, for SOSNet and SIFT the number of keypoints can be reduced to 200 by sacrificing only 2% performance. The trade-off between utility and privacy is shown in Figure 4.6. Overall, we find that SIFT yields the best privacy-utility trade-off among the evaluated descriptor configurations on the Megadepth dataset. We note that these results do not preclude the possibility that other descriptor configurations (i.e., in terms of dimensionality, target dataset, and type) may achieve better results. Ultimately the ideal descriptor chosen will depend on the precise privacy and utility requirements necessitated by the localization service.

4.5.6 Selective Suppression of Features

Globally reducing image features can reduce the potency of the reconstruction attack, but at the same time it reduces the matching accuracy. In this section, we investigate to what extent an object detector can help implement a more selective approach. We identify and mark the sensitive regions in the images using the bounding boxes produced by the YOLO v3 [16] object detector. Based on the bounding boxes, we then suppress any features in these regions. Finally, we apply our reverse-engineering attack and measure the detectable semantic information content in the images before and after reverse engineering (Table 4.3).

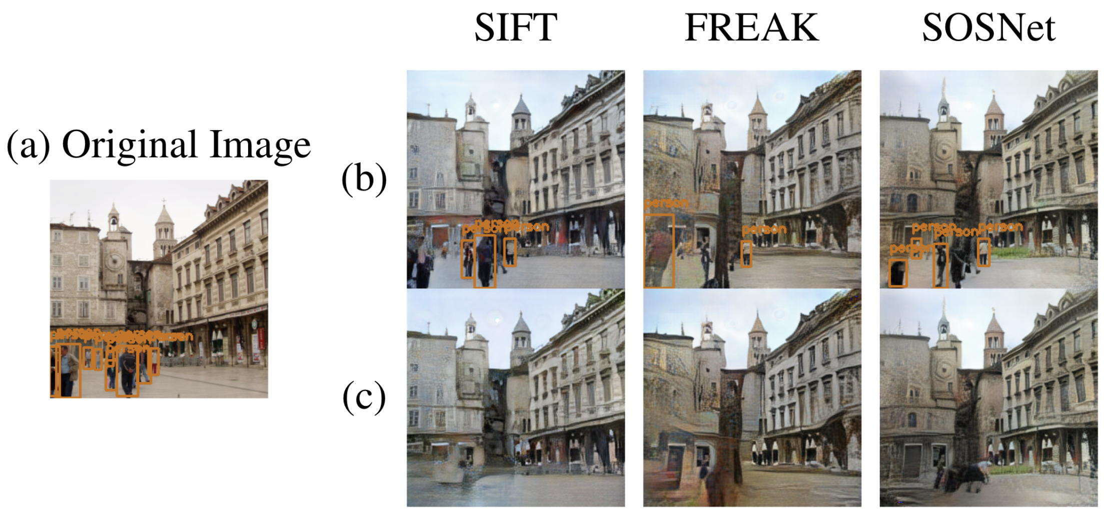


Figure 4.7: **Reverse Engineering after Selective Feature Suppression.** (a) Object detection on original image (b) Object detection on reverse-engineered images (max. 1000 keypoints) (c) Object detection on reverse-engineered images with feature suppression. All objects detected by the object detector without suppression are successfully removed with suppression.

Figure 4.7 shows a qualitative example of how selective feature suppression effectively defeats the object detector; the people detected in the original image do not appear nor are identifiable by the object detector in the reconstructed images. These results confirm our intuition that selective suppression can effectively preserve the privacy around a potentially sensitive region of interest (in our case semantic content of people in the image). Note that the quality of the overall image outside of the marked sensitive regions remains largely unaffected. Finally, the results show that features of private objects should not be shared in order to mitigate privacy risks posed by reverse engineering attacks.

Results for the privacy-utility trade-off of the suppression are given in Table 4.3. Under the evaluated experimental conditions, SIFT and SOSNet give better trade-offs than FREAK; these trends are consistent with the results from Section 4.5.5. Notably for SIFT the utility drops slightly, while the detected objects are almost eliminated.

4.6 Related Work

The concept of reverse engineering local features has evolved over recent years as local descriptors play an increasingly important role. Prior work focused primarily on better understanding the image features. Only recently have there been proposals towards leveraging this line of research to understand the privacy implications. Work towards discovering vulnerabilities and mitigating against attacks remains an emerging area of research.

4.6.1 Recovering Images from Feature Vectors

Reconstruction from Sparse Local Features. Weinzaepfel et al. [129] demonstrated the feasibility of reconstructing the input image, given SIFT [135] descriptors and their keypoint locations, by finding and stitching the nearest neighbors in a database of patches. d'Angelo et al. [130] cast the reconstruction problem as regularized deconvolution problem to recover the image content from binary descriptors, such as FREAK [134] and ORB [155], and their keypoint locations. Kato and Harada [156] showed that it is possible to recover some of the structures of the original image from an aggregation of sparse local descriptors in bag-of-words (BoW) representation, even without keypoint locations. While the quality of reconstructed images from the above methods is far from the original images, they allow clear interpretations of the semantic image content. In this paper, we demonstrate that reverse engineering attacks using CNNs reveal much more image details and quantitatively analyse privacy implications for floating-point [135], binary [134] and machine-learned descriptors [136].

Reconstruction from Dense Feature Maps. Vondrick et al. [157] perform a visualization of HoG [158] features in order to understand its gaps for recognition tasks. To understand what information is captured in CNNs, Mahendran and Vedaldi [159] showed the inversions of CNN feature maps as well as a differentiable version of DenseSIFT [160] and HoG [158] descriptors using gradient descent. Dosovitskiy and Brox [131] took an alternative approach to directly model the inverse of feature extraction for HoG [158], LBP [161] and AlexNet [162] using CNNs, and qualitatively show better reconstruction results than the gradient descent approach [159]. They also show reconstructions from SIFT [135] features using descriptor, keypoint, scale, and orientation information. All the above approaches differ from ours in that we perform the reconstruction from descriptors and keypoints only.

Modern Reverse Engineering Attacks. In the context of 3D point clouds and the AR/VR applications built on top of them, a common formulation of the reverse engineering attack is to synthesize scene views given the 3D reconstruction information. Recent work by Pittaluga et al. [132] showed that it is possible to reconstruct a scene from an arbitrary viewpoint from SfM models using the projected keypoints, sparse RGB values, depth, and descriptors. Our work extends this approach by considering only the modalities available to an attacker as input, which are keypoints and descriptors.

4.6.2 Defences and Mitigations

Mitigations for Attacks on Sparse Local Features. For reverse engineering attacks on local features, one notable recent work [163, 164, 165] proposes using line-based features to obfuscate the precise location of keypoints in the scene to make the reconstruction difficult. The key idea is to lift every keypoint location to a line with a random direction, but passing through the original 2D [164] or 3D keypoints [163]. Since the feature location can be anywhere on a line, this alleviates privacy implications in the standard mapping and localization process. Shibuya et al. [165] later extended this approach for SLAM. Similarly, Dusmanu et al. [166] represent a keypoint location as an affine subspace passing through the original point, as well as augmenting the subspace with adversarial feature samples, which makes it more difficult for an adversary to recover original image content.

Mitigations on Raw Images. Apart from local features, other works try to alleviate the privacy concern around sharing raw images by perturbing the images [167, 168, 81, 169, 170, 171, 172, 173]. One way of achieving this is to mask out or replace the parts of images (e.g., faces) that may contain private information [174, 167, 168]. Another stream of work focuses on encoding schemes or degrading images to prevent recognition of private image content [81, 169, 170, 171, 172, 173]. A few cryptographic methods were proposed to encrypt visual content in a homomorphic way on local devices [175, 176, 177], which allows computing on encrypted data without decrypting. However, such methods are computationally expensive and it is not clear how to apply them to complex applications such as localization.

4.6.3 Relationship to Adversarial Attacks on Neural Networks

Recent work has shown that it is possible to trick deep learning models with adversarial inputs to induce incorrect outputs [178, 179, 180, 181]. For example, an adversarial attack may engineer a perceptually indistinguishable input image to trick a deep learning model into emitting an incorrect classification result.

Conceptually, these adversarial attacks are similar to the defense or mitigation strategies that we will propose, since state-of-the-art reverse engineering attacks on descriptors rely on deep learning models. Our mitigation techniques modify inputs in a way to prevent the deep learning model used in the attack from accomplishing its objective —

reverse engineering the image. However, unlike prior work in this space, our work lifts the insight that inputs can be modified to induce incorrect outputs and leverages it to **defend** against reverse engineering attacks instead of as an attack vector.

4.7 Conclusion

Our work has formulated a privacy threat model to scope the threats to descriptor-based localization. In contrast to prior work, for the first time, we have shown a reverse engineering attack that operates in the real-world scenario, where only sparse local features are available to an honest-but-curious adversary. We found that our reverse engineering attack could reconstruct the original image with surprisingly good quality. We then investigated two mitigation techniques and showed a trade-off between privacy and utility (measured by feature matching). We found that using an object detector to suppress objects slightly reduces matching accuracy (as a proxy for localization accuracy) but gives better privacy results (fewer reidentifiable objects). Finally, our analysis has shown that, among the descriptors and we evaluate, the best overall privacy-utility trade-off can be achieved with SIFT, when compared to FREAK and SOSNet. Privacy (defined as reidentification risk through reverse engineering attacks as specifically described in this paper) may be preserved with the mitigation techniques described in this paper. Looking forward, our work provides initial experiments on some mitigation techniques the community may consider to further the privacy-aware descriptor-based applications research.

Chapter 5

A Privacy-Enhancing Architecture for Crowd Sourced Data

In this chapter, We target biometric information collected from wearable devices which can yield new insights with the potential to improve the health and wellness of those individuals under measurement. The aggregation of this data over ever larger groups compounds this potential benefit by helping professionals understand the full shape of the distribution, however the nature of such biometric data is extremely personal. Prior work has shown how such shared data can leak your gender, age, habits, and can even be linked back to identity. Future computer architectures have a role to play in protecting user privacy, and we find their use in addressing the privacy loss associated with sharing time-series data specifically (such as those collected from wearables) to be most critical. We introduce two privacy-enhancing interventions that, through a small-footprint hardware extension, can both bound the amount of information leaving a user's wearable device and provide differential privacy guarantees. Through a careful formulation of privacy as an architectural design constraint, the examination of interacting privacy-enhancing parameters, a hardware design and evaluation, and the evaluation of privacy versus utility for a suite of privacy-sensitive applications, we show a flexible and effective privacy framework enabling sharing of streaming sensor data

5.1 Introduction

While it often benefits individuals to learn from crowd-sourced aggregated data, individuals may not be happy revealing their *own* data to a central aggregator. Differential privacy [182] provides a rigorous approach to handling privacy under aggregation and is especially well suited for problems of making queries over aggregated data. A differential privacy mechanism ensures that every single user has *plausible deniability* [52], where there exists another set of data that could produce the same response with the same probability. At a high-level there are two primary sub-types of differential privacy: a global (or centralized) model and the local model.

In the global model, a *trusted* central aggregator collects individual data and applies a differentially private mechanism to it [183]. Typically the differential privacy mechanism is applied exactly once, and only after collecting all the data. However, any aggregator has incredible visibility into the life of those whose data it aggregates, especially if that is raw data coming right from a sensor. Under a typical aggregation system, such as that shown in Figure 5.1, the raw data is still gathered in full – which means there exists a single point in the system where *direct* access to the raw data of thousands of users must be possible, in order to actually apply the global differential privacy mechanism. The aggregator must, then, be a party fully trusted not to abuse or misuse the data provided. Unfortunately, examples of user data being subject to significant breaches of privacy are not at all uncommon [184, 185, 186] and while users may be willing to trust some parties, they are most likely not even *aware* of who is aggregating their data. Once data leaves a device there is little control one can assert about how and where that data is aggregated.

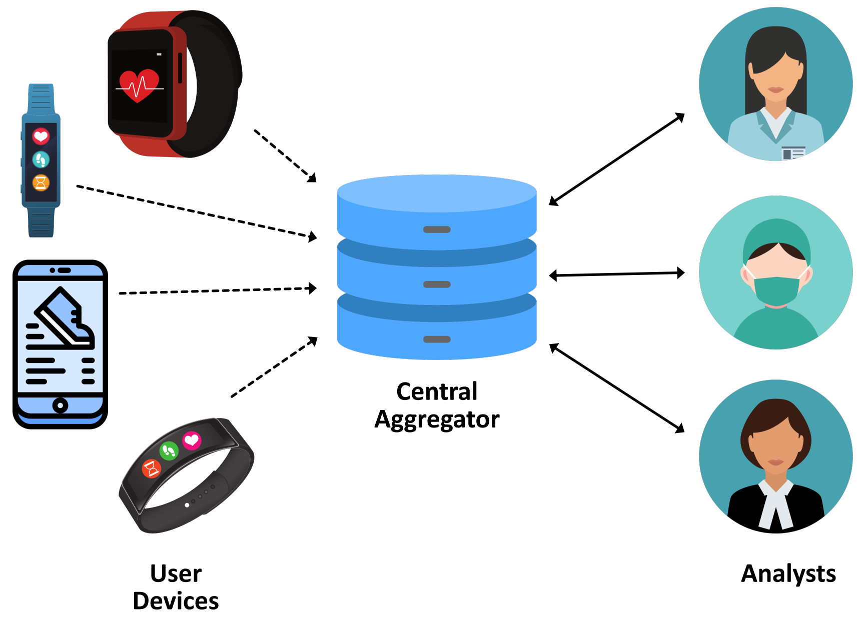


Figure 5.1: Embedded data sources produce continuous streams of data which provide significant value in aggregate form. Traditional models of differential privacy consider what happens as queries are made on centralized collections of that data. Productions can be either local (meaning they are enforced at the source) or global (meaning the are enforced on the collection as a whole).

In contrast, under a local model of differential privacy, the central aggregator collects only *carefully modified* data from individuals, i.e., each individual applies a differential privacy mechanism to their own data [187]. Here the aggregator need not apply any differentially private mechanisms – the privacy of the resulting queries is inherent in the data provided to the aggregator who may view the privacy modifications made as a form of noise. The lack of a single point of failure is a significant benefit and local differential privacy approaches have been adopted by both Google [188] and Apple [189] in recent years for various approaches to sampling.

While differential privacy is generally a useful tool for managing the release of personal data, wearables present some additional constraints because they collect *streams* of personal data through their sensors. When combined, this aggregated information

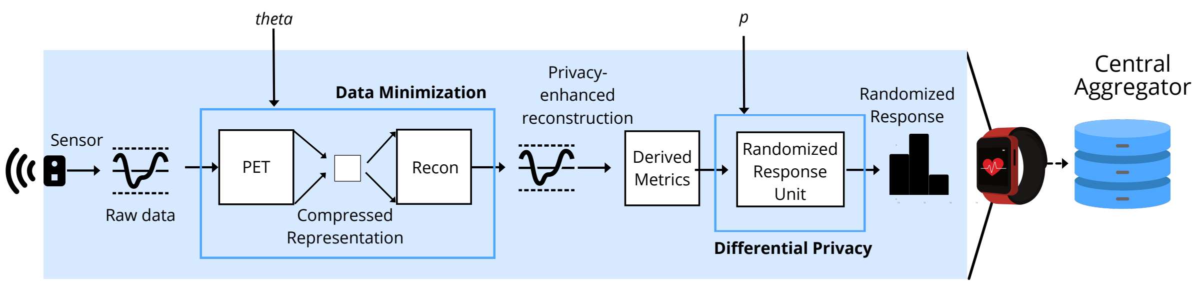


Figure 5.2: System Overview: A key idea of this work is to compose data minimization of sensor datastreams (implemented via a privacy-enhancing transform with parameter θ) with local differential privacy (implemented via a randomized response unit with parameter p) *in hardware on device*, prior to transmission to a centralized aggregator. This protects against intentional and inadvertant (e.g. data breach) violations of the trust boundary at the aggregator. The key insight of our work is that an optimal choice of θ can improve user privacy without affecting the overall utility score of the aggregated information.

can be incredibly helpful to analysts interested in questions that will help assess the overall health of communities, learn about commuting patterns, evaluate infrastructure and resource availability, and much more. However, while this sensor data can be used to monitor individual activities and interactions, the same data can give away smoking habits [190], reveal personal attributes such as age, height, and gender [191], or even lead to user re-identification [192]. Even more problematically, standard differential privacy mechanisms work from the idea of the management of "privacy budget" which, decided *a priori*, limits the amount of information that will be given away. The higher the budget, the lower the degree of privacy protection. When dealing with continuous streams of data, the privacy budget is forced to continuously increase, meaning the release of practically unbounded private information over time [183, 193].

This is a particularly acute problem, and in this chapter we introduce a small privacy enhancement unit that can be integrated directly with sensing hardware to regain control over user data release. Such a unit enables an architecture which *bifurcates* the stream of sensor data into two: one for internal use only and one ready for aggregation. A key challenge of realizing such an architecture is in ensuring that the privacy of the stream of data ready for release is maintained, a challenge we demonstrate can be overcome through the novel integration of information theoretic methods for reduced disclosure and local differential privacy methods. The resulting system is the first privacy-preserving system that couples Short-Term-Fourier-Transform based data minimization and randomized response-based local differential privacy for managing and shaping the release of time-varying data. Interestingly, for different application scenarios different balances of these schemes are optimal, but in all cases the combination works better than any scheme in isolation. A configurable hardware privacy unit allows such a tradeoff to be established for each type of data independently, but at the same time it allows for the establishment of clear and non-bypassable privacy on the data under aggregation. Specifically, the contributions of this work are as follows:

1. We propose a new approach to embedded data privacy in which the architecture produces and manages multiple streams of data, some of which are specifically readied for external aggregation through hardware modification.

2. We demonstrate that stream data across a variety of applications can be most effectively readied for aggregation through a novel combination of short-term Fourier analysis and randomized response.

3. We further show it is possible to embody these techniques in a small hardware unit that works in a completely streaming fashion at low overhead compared to power budget of typical wearable processors.

4. We quantitatively evaluate our approach across motion sensor and ECG data using information theoretic tools from the privacy community and qualitatively against our chosen privacy threat model.

The rest of the chapter is organized as follows. We begin by clarifying our Privacy Threat Model is Section 5.2 before we dive into the high level architecture and ways of measuring success in Section 5.3. From there we describe the algorithms (Section 5.4) and hardware (Section 5.5) at the heart of our approach. We finish up with a discussion of our evaluation, related work, and conclusions in Sections 5.6, 5.7, and 5.8 respectively.

5.2 Privacy Threat Model

Just as an architecture security paper should be clear about the specific threat model it attempts to address, best practice in privacy research is to be clear about a *privacy threat model*. What data is being protected, from whom is it being protected, what types of steps might we assume someone interested in breaching privacy would be willing to take, and under what conditions might we say that privacy has been breached might all be considered of such a model. The strongest class of assumptions one might make are *adversarial* and reduce to *information flow*, such that any leakage of information about a user, given any computational actions by an adversary, could be considered a violation of privacy. However, this model precludes any useful sharing of data. Instead most privacy research assume users are willing to give up *some information* as long as that information cannot, in some sense, be "used against them" as long as some *utility* is derived from that sharing. Specifically we consider systems that tackle the problem of aggregating sensitive, personal data from thousands of users for research and analysis. The goal is to accomplish this while keeping the individual users' sensitive information private to the highest degree possible.

LINDDUN [14] provides an intellectual privacy threat modeling framework and is a mnemonic for the privacy threat categories it supports: (a) Linkability, (b) Identifiability, (c) Non-repudiation, (d) Detectability, (e) Disclosure of Information, (f) Unawareness, and (g) Non-compliance. In Table 5.1, we provide a LINDDUN mapping for our system, which looks at how the untrusted entities and processes interact and what they can learn about the individual when no privacy guards are in place. From our assumptions, we do not consider linkability, and also assume that the individual users already know that their personal data is being used. We also consider managing non-compliance to be out of the scope of this work.

In our setting there are three components: (1) the individual users whose data is being aggregated, (2) the aggregator who is collecting data, and (3) analysts who want to infer trends and patterns among many individuals. The aggregator is considered to be honest-but-curious and follows protocol; sending queries from analysts to the individuals in a timely manner. But, they may try to exploit the information available to them. The aggregator does not collude with individuals, nor the analysts, but have direct access to both parties. Analysts are considered potentially malicious; they may try to learn more information about the individual users by leveraging publicly available data for such inference, building user profiles, and trying to remove noise from the query responses. Security vulnerabilities and client malfunction/misconfiguration that allow direct access to private data through unintended channels, while important, are considered by other research and outside the scope of this work.

Under this setting a trade-off between local privacy and aggregate utility is inevitable. The goal is to strengthen the privacy of the users; more privacy for the individual results in less reliable and useful data for analysts. To this end, we present a software framework and hardware implementation that work together to manage privacy and utility in data aggregation systems that specifically deal with sensitive time-series data collected from wearables.

Table 5.1: Eliciting threats to privacy using the LINDDUN knowledge base. We map the untrusted elements to a privacy threat model described by LINDDUN.

Untrusted Element	L	I	N	D	D	U	N
Aggregator	-	y	n	y	y	-	-
Analyst	-	y	n	y	y	-	-

5.3 Privacy-Enhancing Architecture

In this chapter, we present a privacy architecture capable of safely aggregating streams of personal and sensitive biometric data from users. Our technique includes novel privacy-enhancing algorithms and their low-overhead hardware implementations. Because previous approaches based on local differential privacy alone are insufficient when periodically gathering time-series data from the same device [189], we choose a hybrid approach that combines information theoretic and differential privacy. The key idea underpinning this approach is creation of two different streams of data, one that is strictly to be kept on the device, and one that can be released for aggregation.

5.3.1 A Two Stream Model of Privacy

Modern sensors provide high resolution data that capture an extraordinary amount of information about their users' movements, environment, and interactions. And while that data is useful for providing features users care about (e.g. good accelerometer data is useful for understanding the orientation of a mobile device), that raw data also has a great deal of other information embedded in it (e.g. turning it back into location data). One simple approach to reducing the amount of privacy lost would be to simply reduce the fidelity of the sensors to only provide the amount of resolution needed for a specific task. However this has several problems. First, it requires that we have a clear way of specifying the needs of the task that can be trusted. Second it says nothing about what happens to the data after it is used for the task (e.g. that it is not then shared externally). In reality, there is likely always a local use for a sensor running at its highest sensitivity or otherwise the sensor in question would not provide sufficient value to merit inclusion on the device at all. Such a scheme either over-limits the local capability of the device out of an abundance of caution or under-protects the privacy of the users. A second solution is to trust the local applications to not abuse the sensor data provided to them and to properly aggregate that data responsibly through calls to some privacy preserving framework. Such a framework might even be local, with software hooks in place to downgrade information before aggregation. However, exactly where those hooks must be invoked and understanding how that data is allowed to flow through the system is not an easy problem.

A related problem comes up in the development of high-assurance systems when attempting to enforce access control. In that community a simple but powerful concept is that of a "reference monitor". A reference monitor has the job of inspecting every access on the system, traditionally a read or write reference, and examining it for compliance with an access control policy. Allowed accesses are completed as requested and disallowed accesses are simply rejected. Four critical properties must hold for a reference monitor system to be secure: it must by non-bypassable, it must be verifiable, it must always be invoked, and it must be tamper-proof. A privacy scheme requiring minimal trust would ensure all data is transformed into a privacy-enhanced form before being given "access" to the outside world – and one would hope that those same four properties would hold for whatever mechanism is responsible for modifying the data.

Here we believe that architecture has an important role to play. Hardware is much harder to modify than software, it can be more easily designed with a restricted set of possible behaviors, and different physical interfaces can provide true separation of information. In the case of privacy this is extremely helpful because it allows us to separate some levels of sensor data for internal use and other levels of data for aggregation. If data flows directly from the sensor to privacy-enhancing transformations and both are directly implemented in hardware, it is trivial to ensure non-bypassability to the interface. At that point it becomes a standard information flow management problem to ensure that only privacy-transformed data can exit the system, a problem well studied in numerous prior works [194, 195, 196]. The more interesting part of the problem then is how would one actually enhance the privacy of the sensor data to prepare it for aggregation?

Our approach there is two-fold. First, we can reduce the total disclosure of sensor data by producing a privacy-enhanced reconstruction of raw sensor data from a lossy compressed representation. Specifically we propose a new privacy-enhancing transform (PET) unit that is parameterized by θ, the disclosure minimization rate which controls, in a sense, how "lossy" the compressed representation will be. A full-length privacy-enhanced signal stream is reconstructed from the compressed representation, and used to obtain derived metrics (e.g. step counts from accelerometer signals, beats per minute from ECG signals, etc.) Second, all of the queries we consider in this work are designed such that the values of derived metrics over n time units can be encoded as histograms, and a randomized response unit further strengthens the privacy of the response by adding noise through differential privacy mechanisms. This novel combination of reduced disclosure and random response in fact works far better than either approach independently, and yet the resulting hardware for such a transform is small, only 0.134mm^2 on 7nm technology as shown in Section 5.6.3. A graphical depiction of the approach can be found in Figure 5.2.

While we describe the specific hardware required later in Section 5.5, central to the design process is the idea of finding an approach that either "reduces" the loss of privacy for a given utility, or "improves" utility under an given degree of privacy. To accomplish either of these we require a way of quantifying these dimensions of design, and while no measure will capture every aspect of privacy or utility, there are several insights from the privacy community one can draw upon.

5.3.2 Quantifying Utility

A metric of utility should attempt to capture the degree of *usefulness* of the aggregated data. While in the general case putting a number to how useful something is is no easy task, here we specifically care about how useful some data is *in comparison* to the case where complete sensor data is available. One task-independent measure then of utility is how close a privacy-enhanced reconstruction of some data is to this ground truth. Of course any single measure of closeness may hide pathological cases that just happen to be similar in some shallow way, so we consider 4 different utility functions defined below.

1. Utility Score and Utility Score with Margins. If analysts are interested in asking queries in the style of "what is the bin with the highest counts?", the utility metric must capture how the normalized bins are ranked by counts. We introduce two utility scores as follows.

$$\text{ranking\_score}(t, r) = \sum_{i=1}^{|t|} \text{int}(\tilde{t}[i] == \tilde{r}[i]) \tag{5.1}$$

   where $\tilde{t} = \text{argsort}(t)$, and $\tilde{r} = \text{argsort}(r)$. Similarly we define the utility score with margins as:

$$\text{ranking\_score}(t, r) = \sum_{i=1}^{|t|} \text{int}(\gamma_\ell \tilde{t}[i] <= \tilde{r}[i] <= \gamma_h \tilde{t}[i]) \tag{5.2}$$

   where γ_ℓ, γ_h define a margin of error in matching, e.g. $(\gamma_\ell, \gamma_h) = (0.9, 1.1)$ to indicate a 10% margin.

2. Mean Absolute Error and Mean Relative Error We may also quantify utility by computing the mean absolute error (MAE) and mean relative error (MRE) between the normalized histograms.

$$\text{MAE}(t, r) = \frac{1}{|t|} \sum_i |t[i] - r[i]| \tag{5.3}$$

$$\text{MRE}(t, r) = \frac{1}{|t|} \sum_i \frac{|t[i] - r[i]|}{t[i]} \tag{5.4}$$

where $t[i]$ and $r[i]$) represent the aggregated value of ground truth histograms and the aggregated values of predicted (privacy-enhanced) histograms respectively.

5.3.3 Quantifying Privacy

For two random variables T, R (e.g. representing the true and response value of some wearable metric), the mutual information (or information gain) between them is defined as:

$$I(X; Y) = D_{\text{KL}}(P_{(T,R)} \| P_T \otimes P_R) \tag{5.5}$$

where D_{KL} is the Kullback–Leibler divergence [197], and T, R are summarized by the probability distributions P_R and P_T. In the framework used in this chapter, we have a natural discrete formulation for P_T and P_R as histograms generated from the distribution of derived sensor metrics corresponding to unfiltered/unperturbed sensor data, and data minimized and perturbed sensor data, respectively. Following the literature in the field [198, 199], we define a local measure of privacy in which quantify the difference between these discrete M-bin distributions using the Kullback–Leibler (KL) divergence:

$$D_{\mathrm{KL}}(P \parallel R) = -\sum_{i}^{M} P[x] \log_e \left(\frac{R[i]}{P[i]} \right) \tag{5.6}$$

Due to the use of the natural logarithm in Eq. 5.6, this privacy measure is interpreted in *nats* taking values $\in (0, \infty)$, indicating maximally identical (bad privacy) and independent (good privacy) distributions for T and R.

5.4 Private Stream Generation

As mentioned above, a key insight of our work is that, after spliting the streams of data into one for local-use and one for aggregation, we apply privacy-preserving algorithms directly in hardware. This hardware can be implemented locally (potentially even directly on the sensor hardware itself). In demonstrating this concept we use two specific techniques that complement each other in the enforcement of privacy.

First, we apply a novel privacy-enhancing transformation that (a) strengthens the approach in the face of multiple query-response cycles, and (b) addresses identifiability, detectability, and disclosure of information through reduced disclosure. The main idea is to make use of a Short-Term Fourier Transform (STFT) to reduce the information shared to the bare minimum required to get the job done. Exploiting structure inherent to the data, we can reduce the amount of information that is shared in a way that the utility falls off gracefully with the degree of reduction. From this reduced set, we can then apply local differential privacy through randomized response in a way that further addresses identifiability, detectability, and ensures non-repudiation. Applying these privacy enhancements (reduced disclosure and randomized response) changes the aggregated sensor data values. While the ideal scenario is to maximize both privacy and utility, in reality there is a fundamental tradeoff between the two that we need to navigate. However, as we will show in the Section 5.6, each of these two approaches adds something unique to the solution.

5.4.1 Privacy-enhancing STFT

Spectral analysis of time-series data has numerous applications including signal artifact removal, event detection, and compression over bandwidth-limited communication channels. For digitally-sampled signals, the Discrete Fourier Transform (DFT) *transforms* a length-N vector $x \in \mathbb{C}^n$ into a length-N vector of frequency coefficients $X \in \mathbb{C}^n$, as $X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn}$, which affords the simple matrix formulation $X = Fx$. Since F is a unitary transform, we can recover the signal x exactly without information loss using the inverse transform $x = F^{-1}X = F^T X$. The Fast Fourier Transform (FFT) is a computationally efficient way to implement the DFT (or inverse DFT) by leveraging a factorization of $F \in \mathbb{C}^{n,n}$ into a product of sparse (mostly zero) factors, reducing the complexity of computing the DFT from $O(N)$ to $O(N \log N)$. There are many FFT factorization and transform implementations, but the utility of each depends on the exact value of N.

For streaming sensor data, the discrete-time Short-Time Fourier Transform (STFT) is often used instead of the FFT because it provides frequency decomposition of local segments of a time-domain signal. This desirable for two reasons: (1) local frequency analysis enables higher *temporal resolution* for event detection and communication signals, and (2) this combats the rising power and complexity cost of using higher sequence lengths. Specifically, the STFT may be defined as:

$$X_{\text{STFT}}[m, n] = \sum_{k=0}^{L-1} x[k]g[k-m]e^{-j2\pi nk/L} \tag{5.7}$$

and the inverse STFT may be defined as:

$$x[k] = \sum_{m} \sum_{n} X_{\text{STFT}}[m, n] g[k - m] e^{j2\pi nk/L} \tag{5.8}$$

where $x[k]$ denotes a signal indexed by k and $g[k]$ denotes an L-point window function. Thus, the STFT can be interpreted as the rolling L-point DFT of the product $x[k]g[k-m]$, where g is responsible for selecting and filtering a local segment of x. The shift between each window m can be chosen to achieve the desired temporal resolution, but is normally chosen to be $\geq \lceil L/2 \rceil$ to support lossless reconstruction of x. The window function g and length are typically chosen to maximize the frequency resolution and mitigate high-frequency artifacts arising from clipping x to a finite-length window (Fig. 5.3).

There have been various ways devised to implement FFT in hardware and the most common is to use pipelined butterfly-based architectures [200]. In order to construct an STFT module in hardware, FFT units are typically reused along with some windowing and data reordering units to accommodate different window sizes and overlap percentages. In this work, in order to simplify our design and because of the nature of the data we work on, we use a fixed windows size and overlap percentage as described in Section 5.5.

Specifically, we use a one-sided FFT-based STFT algorithm that not only affords a relatively simple hardware implementation, but also enables a novel privacy-enhancing transform close to the sensor. In particular, for L-sample time-window we perform a memoryless privacy-enhancing transform W after the application of the L-point FFT, expressed as (Fig. 5.4):

$$X_{\text{PP-STFT}}[m] = \text{ReLU}(W_\theta X_{\text{STFT}}[m] + b_\theta) \tag{5.9}$$

where $X_{\text{STFT}}[m]$ is the output of the FFT-unit for a given time segment, and W_θ and b_θ

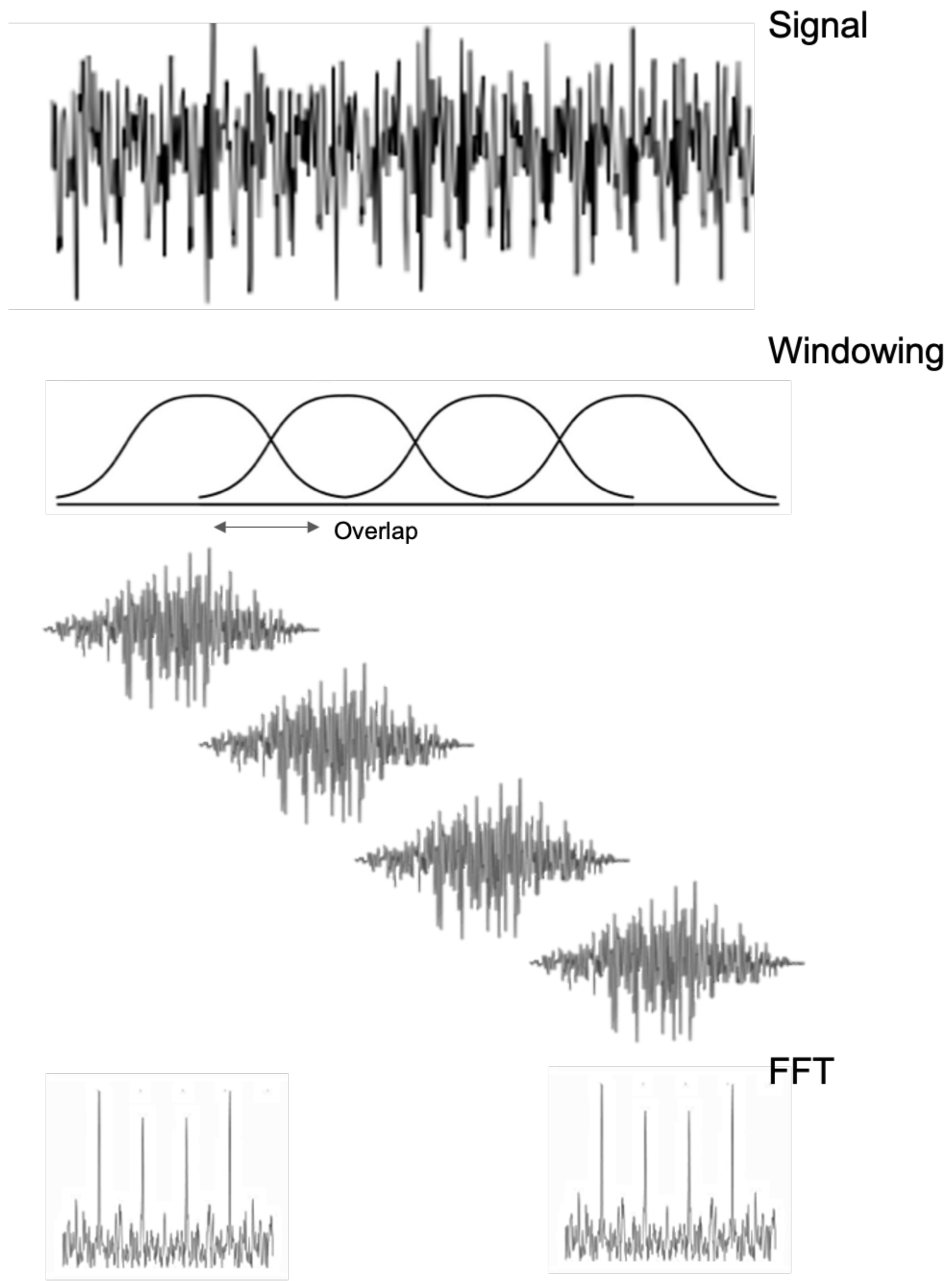


Figure 5.3: The STFT filters and transforms data using a sliding window, resulting in a time series of spectral coefficients. There poses a fundamental tradeoff between frequency resolution and time resolution in the spectral coefficients, but still affords perfect reconstruction when utilizing all the data.

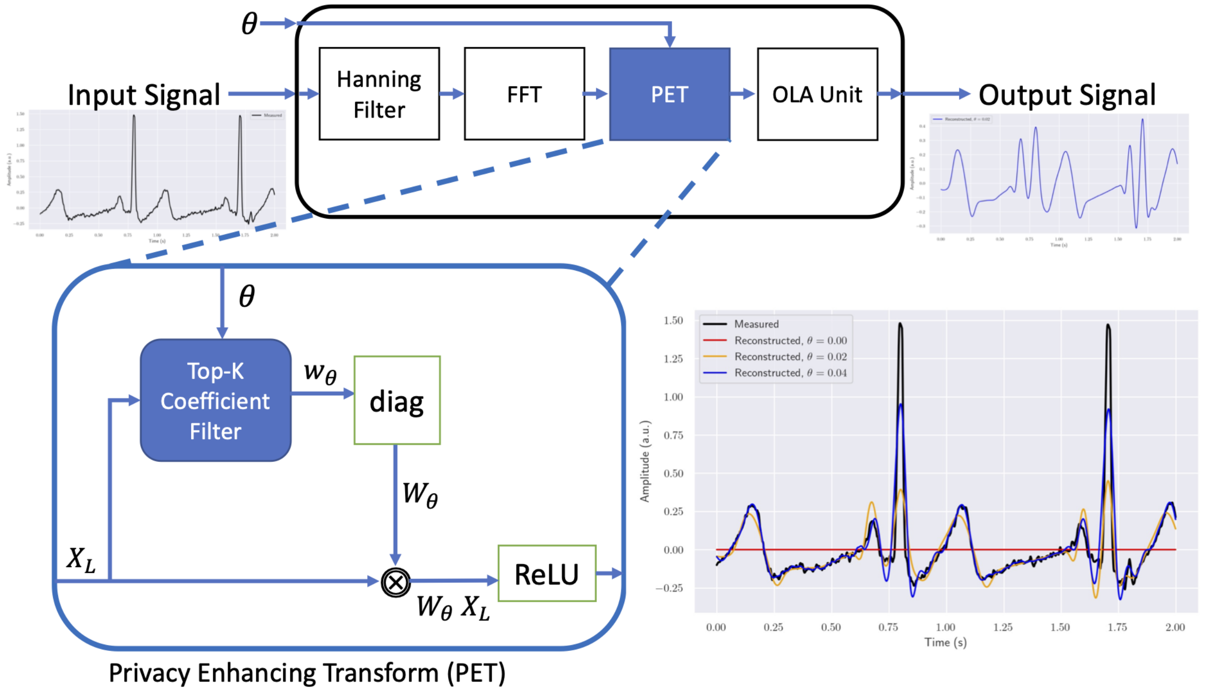


Figure 5.4: An adaptive privacy-enhancing transform (PET) embedded in the STFT Module. It is parameterized by the privacy parameter θ and adapts W_θ to the signal content of each time segment. Without additional compression, the bit leakage upperbound scales linearly with θ for each time window.

are parameters of the transform.

The optimal choice of W_θ and b_θ depends largely on the application, and should typically be chosen via offline optimization to maximize utility within the user's privacy budget. Of primary importance to this chapter is the fact that W_θ should be parameterized by a user-controllable privacy parameter θ, although it can more generally be a function of the signal content as well, as $W_\theta = W(\theta, x)$. In lieu of a large, centralized, offline optimization framework here we use a simple signal activity-dependent diagonal parameterizations of W based on signal energy content and number of shareable bits in the user's privacy budget (Alg. 1) that zeros all but the top $K = \text{floor}(L \cdot \theta)$ Fourier coefficients with the highest magnitudes for each time segment. That is, θ is interpreted as a fraction of the number of coefficients to retain, which scales linearly with the number of bits shared. Another simple approach (not shown here) is to interpret θ as a fraction of total information content to retain at each time segment. Once the transformed coefficient vector for a given time segment $X_{\text{STFT}}[m]$ is computed, it is buffered for the inverse STFT (ISTFT) computation, e.g. using the classical overlap-add (OLA) technique [201].

Algorithm 1 Top-K Coefficient Filter

```
function FILTER_TOPK($X_L$ , $\theta$)
    $K \leftarrow \text{floor}(L \cdot \theta)$
    $X_{\text{sorted}}, I_{\text{sorted}} = \text{sort}(abs(X_L))$
    $I_{\text{keep}} = I_{\text{sorted}}[1 : K]$
    $w \leftarrow \{0\}_L$
    $w\left[I_{\text{keep}}\right] \leftarrow X_L\left[I_{\text{keep}}\right]$
    $W \leftarrow \text{diag}(w)$
    return ReLU($W \cdot X_L + b$)
end function
```

5.4.2 Local Differential Privacy through Randomized Response

Randomized response is a mechanism for local differential privacy. First proposed as a surveying technique for asking sensitive questions [202], randomized response provides the surveyors with *plausible deniability* [52]. For a given sensitive question, the respondent first flips a fair coin in secret, and answers "Yes" if it comes up heads, or truthfully otherwise. This interaction is shown in Figure 5.5(a). A later variant [183] provides deniability for both "Yes" and "No" responses: the respondent flips a coin in secret, and answers truthfully if the coin is heads, or flips another coin. If the second coin is heads, the respondent answers "Yes" and "No" otherwise. This randomized response in Figure5.5(b) provides deniability for both "Yes" and "No" responses and satisfies differential privacy for $\epsilon = 1.09$.

This formulation works well for Yes/No questions but if the query is more complex,

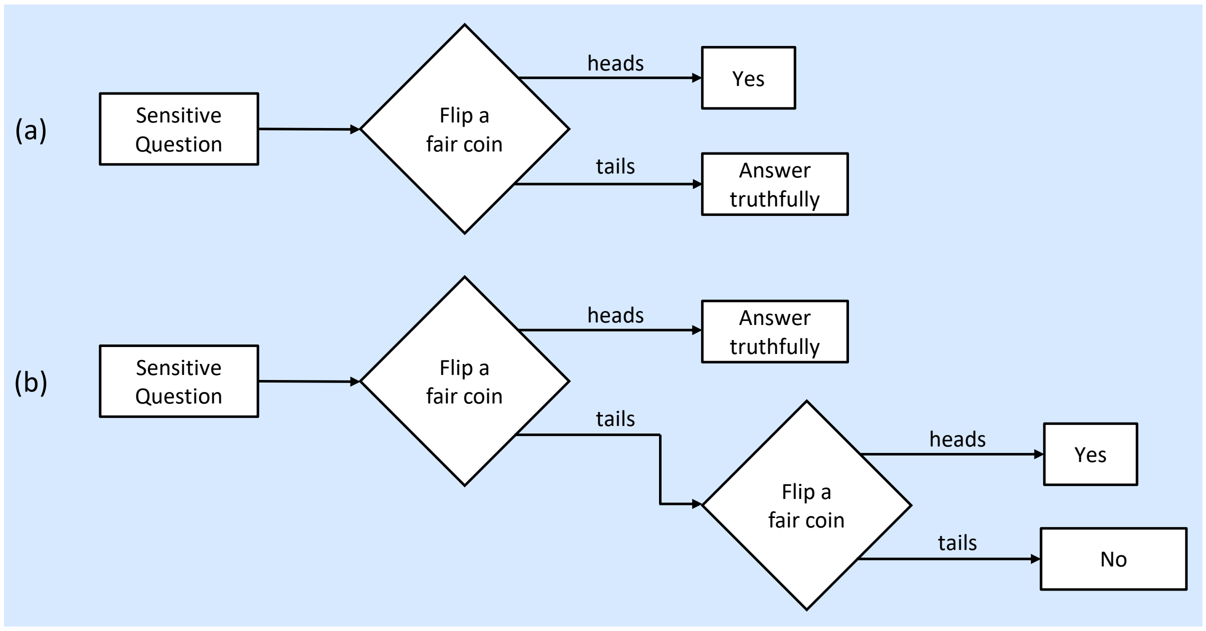


Figure 5.5: (a) Randomized response, (b) Differentially Private Randomized Response with parameter p can be used to provides deniability for both "yes" and "no" responses when answering sensitive survey questions.

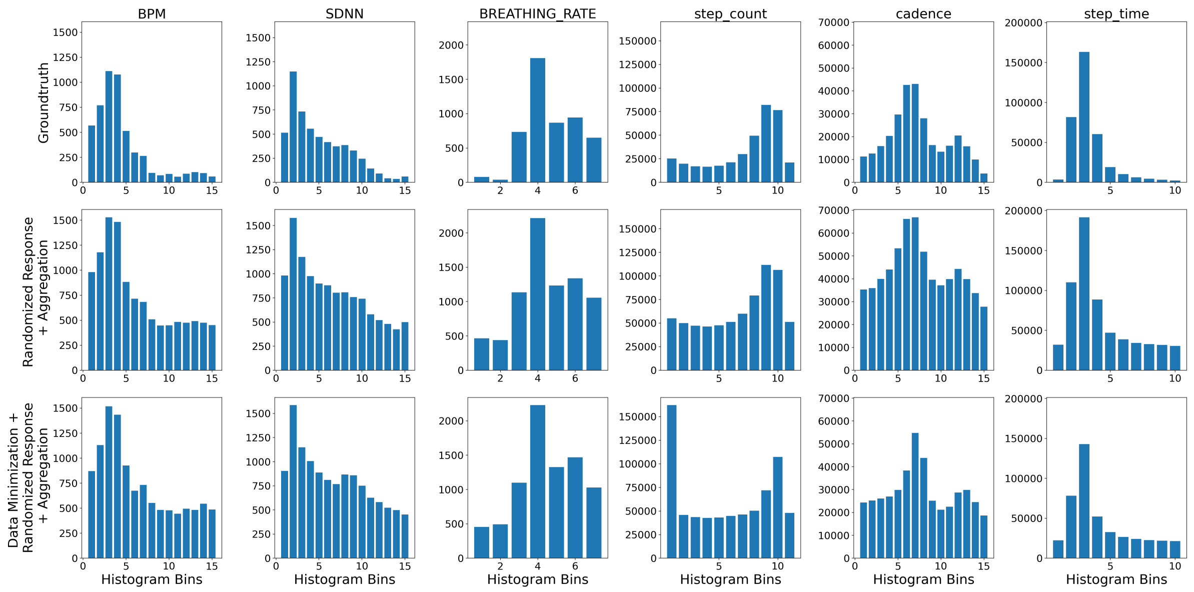


Figure 5.6: **Aggregated results for $\theta = 0.03$, $p = 0.9$.** Row I is the ground truth aggregated histogram, Row II aggregates over locally randomized responses (RR), and Row III aggregates over locally randomized responses over reduced disclosure signals (DM + RR)

e.g., responding with a histogram of counts, we use the unary encoding (also used by RAPPOR [188]). This requires two steps that are performed locally:

- *encode*, which one-hot encodes the response, and
- *perturb*, which perturbs the encoded response

Perturb flips bits to ensure local differential privacy based on parameters p and q.

$$\Pr[B'[i] = 1] = \begin{cases} p & B[i] = 1 \\ q & B[i] = 0 \end{cases} \tag{5.10}$$

Given p and q, the noise budget is given by:

$$\epsilon = \log\left(\frac{p(1-q)}{(1-p)q}\right) \tag{5.11}$$

For $p = 0.75$ and $q = 0.25$, $\epsilon = 2.19$. ϵ measures how much the risk to an individual's privacy may increase due to that individual's data being included. **A higher ϵ means less privacy protection** and increase to the privacy risk is proportional to $exp(\epsilon)$.

The *aggregation* takes into account the number of bit perturbations in each category, which is a function of p, q, and number of responses, n,

$$A[i] = \frac{\sum_j B'_j[i] - nq}{p - q} \tag{5.12}$$

Another serious difficulty when dealing with time-series sensor data from wearables is that local differential privacy may not provide strong (or any) privacy guarantees when collating multiple rounds of differentially private data [189], potentially leaking unbounded information. Typically, a privacy budget is decided beforehand to apply over the data to guarantee LDP. The higher the budget, the lower are its privacy protections.

When collecting multiple LDP data, due to composition theorems [183, 193], the privacy budget continues to increase, thereby failing to protect the data. Unfortunately, many current and state-of-the-art systems [188, 203] fail to address this concern. Alternative strategies such as those based on information theory [204, 4, 7] can be useful for transforming time-series data to leak less information. These frameworks often consider the mutual information between the aggregated data and latent information that can be inferred from it as the measure of privacy. While there is no additional noising mechanisms added to these methods, they rely instead, on removal of information, or data minimization to protect privacy.

When considering time-series sensor data, an individual's data contribution is rarely limited to one single differentially private transaction. In fact, multiple transanctions over streaming windows are inherent to time-series sensor data [205, 206]. But this stream of (individually) differentially private data incurs an overall loss of sum of privacy losses of each transaction [183, 193], making it exceptionally important to consider not only individual transactions, but the number of transactions per time period over the lifetime of the data-sharing as well. While prior work has shown that this lifetime leakage may be unbounded [189] for certain uses, many current and state-of-the-art systems have failed to address this issue. In this chapter, we present information theoretic techniques to minimize this unbounded loss that current techniques have yet to address.

The input to the Randomized Response Unit (RRU) is the output of the reconstructed output signal from the Privacy-Enhancing Transform (PET) Unit described previously.

Once the stream for these queries is answered, we can begin the process of applying randomized response to them. As discussed above, we begin by encoding our signal. The encoding we use for the unary mechanism is one-hot encoding.

Consider the example query that ISCA 2022 organizers might ask, in order to determine how walkable New York City is: *"What is the distributions of steps taken per day by* *individuals living in the city of New York?"*. When querying this data, the analysts also present the clients with the domain $d = \{[0-1000], (1000-2000], (2000-3000], (3000-4000], (4000-5000], (5000, \infty)\}$.

At each client device, the encode unit will accurately answer this and provide a one-hot encoded response; one of the encoded responses may appear as so: $encoded\_response = \{0, 0, 0, 1, 0, 0\}$. The perturb unit, functioning based on Eq. 5.10 might respond in the following way: $perturbed\_response = \{0, 1, 0, 1, 1, 0\}$.

This *perturbed_response* is collected across all the client devices and its response is based on the privacy budget parameter ϵ. The aggregator further interprets *perturbed_response* and stores it in its database before responding to the analyst with the final distribution.

Figure 5.6 presents the aggregation of derived data metrics across all the stream devices. For each metric, we present three histograms: the ground truth aggregate (Row I), the aggregate over randomized responses (Row II), and the aggregate over privacy-enhanced randomized response (Row III). The x and y axes for all subplots are bins and counts respectively. The ground truth is trivially computed by summing true histograms drawn from untouched sensor data across all devices. The randomized histogram response drawn from untouched sensor data with ($p = 0.9$) across all devices presents a differentially private histogram. Note that while the relative "shape" of the histogram remains similar to the ground truth, the counts are significantly different due to the `encode` and `perturb` functions applied during randomized response. Row III presents the aggregated histograms over randomized responses collected after applying the privacy-enhanced transform over the sensor data with $\theta = 0.03$. Here we see a change in both the "shape" and counts.

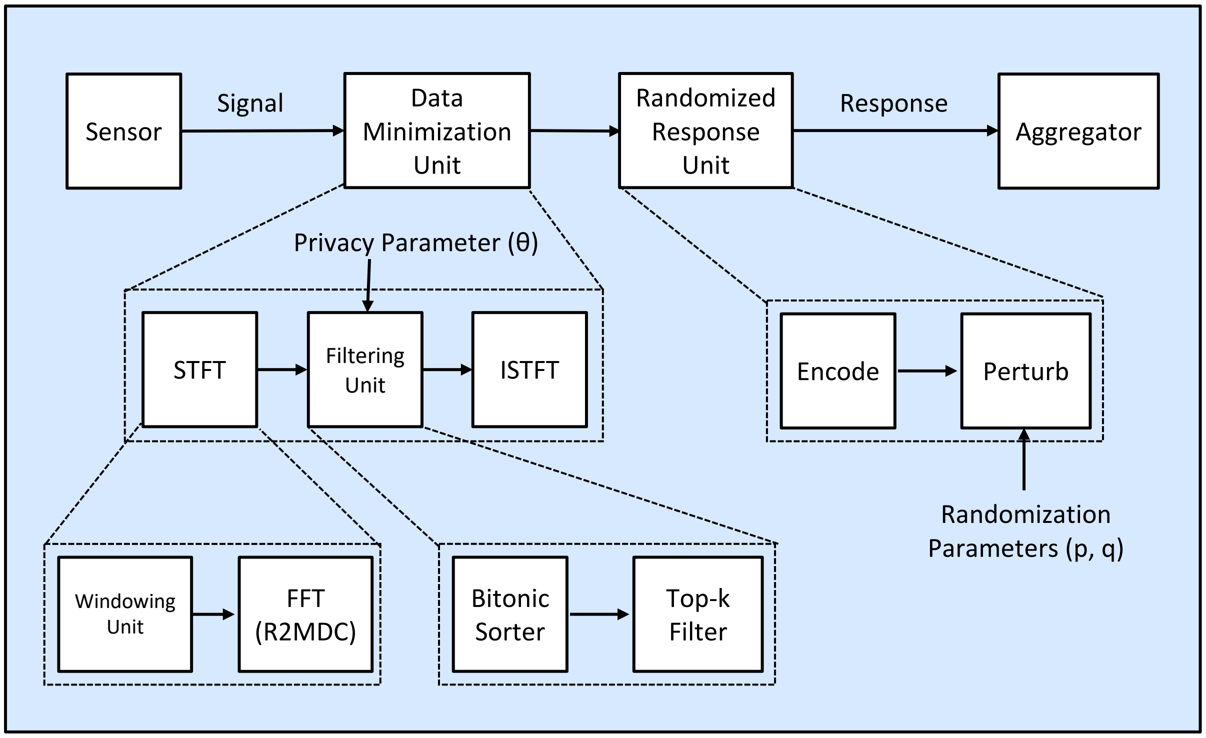


Figure 5.7: **Overview of System Components.** Our proposed Data Minimization Unit (DMU) and Randomized Response Unit (RRU). The DMU contains the STFT and Filtering modules while RRU has Encode and Perturb modules.

5.5 Hardware Architecture

In this section we describe the design of the privacy enhancement stream of the two stream model. Fig.5.7 shows the overview of the system we propose. It consists of Sensor, Aggregator, and the two units we propose - Data Minimization Unit (DMU) and Randomized Response Unit (RRU). As we see in Figure 5.7, there is an notion of nonbypassability built-in to our implementation.

5.5.1 Data Minimization Unit (DMU)

Figure 5.7 shows the STFT-based Privacy-Enhancing Transform architecture. This unit consist of the input/reconstruction buffers, the STFT/ISTFR units, and the Filter unit implementing the Top-k filtering. For the STFT unit, since we are using a fixed 50% overlap window, we adapt a simplified hardware consisting of a windowing unit and a 256-point R2MDC (Radix-2 Multi-path Delay Commutator) [200] FFT unit. A multi-path FFT design was chosen over a serial one to optimize for speed and the R2MDC provides a good tradeoff for speed and hardware complexity [200]. The output of the STFT unit is then used as an input to the Filtering unit. In order to implement a Top-k Filter, we used a Bitonic sorter to sort the data stream and to facilitate selection of threshold values which will be used by the Filter to extract Top-k values only (others values in stream are zeroed out in position). The 16-channel Bitonic sorter can be run iteratively in multiple cycles, saving results in buffers in order to save area and power, since performance is not critical with the low data rate. As noted earlier, we only explore Top-K filtering in this work but the idea is applicable to other filtering mechanisms.

5.5.2 Randomized Response Unit (RRU)

Figure 5.7 also shows the Randomized Response Unit. It consists of Encode and Perturb units. The Encode units converts the signal into a bitstring where a bit is set to '1' in the location of the bin. For example for a signal of value 10 where the bin size (uniform delta) is 5, the corresponding bitstring output of the Encode Unit is a 32-bit length bitstring '0x04'. Note that the number of bins of the query is assumed to be a maximum of 32 and that the LSB of the bitstring always contains the lowermost bin. To implement this, we used a divider unit and then a shifter to move '1' depending on which bin the values are located. Finally, once the encoded bitstring is formed, the Perturb unit is used to add noise on each of the bits of the encoded bitstring. For this work, as a demonstration, we simply use an 32-bit LSFR/CASR-based PRNG. The output of this is a "noisy" bitstring that acts a response which is then sent to the aggregator.

5.6 Evaluation

5.6.1 Experimental Setup

We demonstrate our privacy approach on raw data from two types of sensor readings: (1) accelerometer, and (2) electrocardiogram (ECG). The raw accelerometer data is obtained from the ExtraSensory Dataset [207] and contains the magnitude of acceleration in x, y, and z directions over time. The dataset was collected on everyday devices such as sensors from smartphones and smartwatches from users that were engaged in their regular natural behavior. Sampled at 40Hz for just about 20 second durations, the dataset contains $377,000$ streams of accelerometer data. We use this to simulate 5027 personal devices sharing events from 1500-second windows. For the ECG experiments, we use the MIT-BIH Arrythmia Database [208] which contains 1440 minutes of ECG data sampled at 360Hz. After applying standard data augmentation techniques, we use this to simulate approximately 5600 30-second streams spread across 231 devices.

For each device, we first perform data minimization with parameter θ over the entire data stream (multiple 20-second and 30-second measurements for accelerometer and ECG data, respectively), which results in a privacy-enhanced signal stream of identical duration. As mentioned, data minimization is performed using the privacy-enhanced STFT with parameters $L = 256$ and $m = 128$ (Eq. 5.7), which translates to roughly 8 seconds of accelerometer data and 711 milliseconds of electrocardiogram (ECG) data at a time. A one-sided FFT is used because the input accelerometer and ECG sensor data are real-valued, so negative portion of the spectrum is the complex conjugate of the positive half.

We compute **step count**, **cadence**, and **step time** as derived metrics from the reconstructed accelerometer data and **beats per minute (bpm)**, **heart rate variability (HRV-SDNN)**, and **breathing rate** from the reconstructed ECG data, for each time

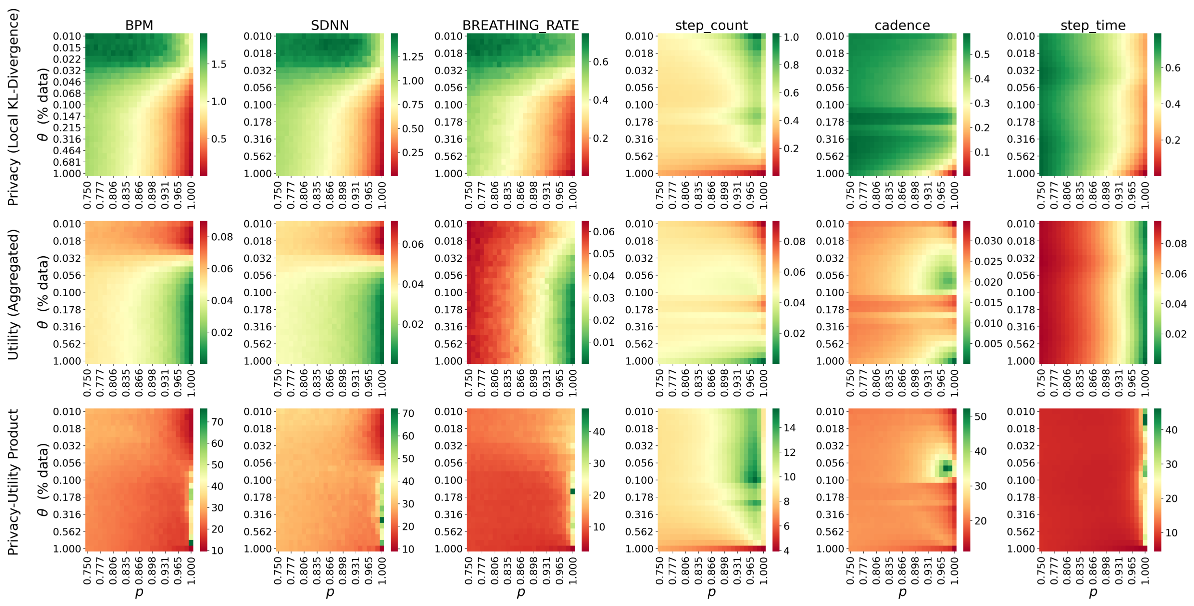


Figure 5.8: **Privacy Parameter Space Exploration:** 2-D histograms, or *heatmaps*, indicating the privacy metric KL-Divergence (Row I), aggregate utility metric Mean Absolute Error (Row II), and Privacy-Utility Composition (elementwise division of Row I and Row II) as a function of privacy parameters θ and p.

segment. The domain for the sensor metrics are shown in Table 5.2. For a single device, we generate a histogram using the metrics computed from the derived data, which is input to a randomized response mechanism with parameter p to produce data ready for aggregation. In our tests, we aggregate query responses from approximately 5000 accelerometer devices and 231 ECG devices, for various values of θ and p, in order to search the privacy-utility tradeoff space. Specifically, for each derived metric we compute and average over all devices the KL-divergence between the groundtruth histogram and released histogram as a measure of privacy, and the MAE, MRE, and ranking scores of the true and aggregated histograms as a measure of utility.

Table 5.2: Sensor data and derived metrics.

Sensor	# Devices	Derived Metric	# Bins for Query
Accelerometer [207] (∼377K samples)	5027	Step Count	11
		Cadence	15
		Step Time	10
ECG [208] (∼5600 samples)	231	BPM	15
		HRV-SDNN	15
		Breathing Rate	7

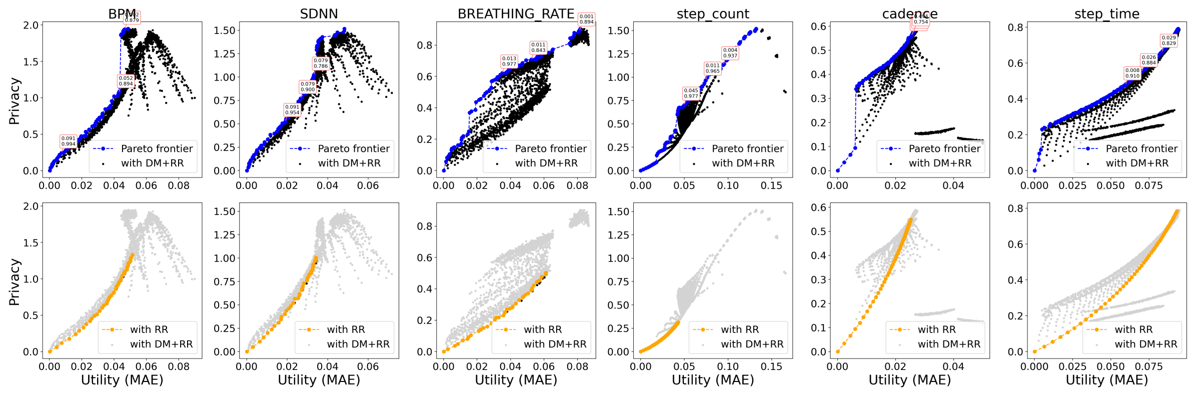


Figure 5.9: **Pareto-optimal Privacy-Utility points:** Row I: Black points mark the full search space ($\bar{\theta} \times \bar{p}$), Blue points mark the resulting Pareto-frontier. Row II: Grey points mark the full search space, Orange points mark optima if only randomized response (RR) is used. Comparison of rows indicates that better optima (upper left is best) can be achieved by composing data minimization (DM) with RR, as indicated by Gray points lying above Orange ones in Row II.

5.6.2 Finding Optimal Privacy-Utility Trade-off Points

While we present the resulting histogram responses from a single θ, p setting in Figure 5.6, to maximize both privacy and utility, it is necessary that we observe how these metrics change as we turn the knob on θ and p values. We take a grid search approach to find the best points and present these results in Figure 5.8, which has six columns for the derived metrics from accelerometer and ECG data. The grid search over the privacy metric (KL-divergence) are shown in Row I and the utility metric (mean absolute error) are shown in Row II. Row III is a composition that maximizes privacy and minimizes error through elementwise division of Rows I and II. We evaluate 50 logspaced $p \in [0.75, 1.0)$ and 50 logspaced $\theta \in [0.01, 1.0]$; these are the x and y axes in each of the subplots. In Row I, the value at each block is represented through its color, `Green` indicating better values than `Red`. We report the average values computed over all the available devices (i.e., 5000+ accelerometer and 250+ ECG devices).

While Figure 5.8 gives us a sense of the search surface we must navigate when setting privacy parameters for dealing with competing privacy and utility objectives, our framework also allows us to navigate the privacy-utility space directly. We show this in Figure 5.9. Each subplot in Row I of this figure is used to find the best possible privacy, given that you want to operate at a certain utility. The x-axis is the utility metric (here we show mean absolute error) and the y-axis is privacy (KL-Divergence) and each point on the scatter plot is a unique privacy-utility point we discovered during the grid search from Figure 5.8. Our goal is to maximize the privacy metric and minimize the utility metric (error), and the best points lie in the top-left corner of the plot. We mark the Pareto frontier in Blue to indicate the best operating points available to us.

Row II of Figure 5.9 demonstrates that it would not have been possible to achieve the Pareto-optimal points **without the novel data minimization technique intro-**

duced in this chapter. The grayed out points are the same as the ones that appear in Row I. The Orange points are privacy-utility points at $\theta = 1.0$, i.e., without any data minimization. As we can see, for each metric there are several gray points above the Orange curve that indicate that correspond to better tradeoff between privacy and utility using data minimization. Data minimization leads to both more discoveries, as well as better discoveries, leading to strictly better and more flexible privacy settings.

5.6.3 Hardware Overhead

To evaluate the hardware overhead of our proposed system, we implemented the RTL of the DMU and RRU. Our R2MDC-based STFT is based on an existing implementation [209]. Synthesis was done using 7nm process technology [210] in Synopsys Design Compiler for 100 MHz operating frequency. Note that the most recent wearable processors are already manufactured in 5nm process [211]. Also, since the nature of the sensor we are interested in (low sampling rate), the hardware could be synthesized in significantly slower frequencies. Table 5.3 shows the breakdown of the main components. The Randomized Response Unit consists of around 17K gates and consumes a power of 100uW while the Data Minimization Unit consist of around 850k gates with roughly 20mW power consumption. The total power overhead of the entire design is small compared to the typical range of power consumption of wearable CPUs which is within 450 mW [212]. Area requirements for a wearable SoCs such as Qualcomm's Snapdragon 400 are in the range of 50mm^2 with 32nm technology [212]. If we scale our design as described in [213], then it occupies around 6.27mm^2.

Table 5.3: Hardware Overhead Results. Figures were collected from synthesis of implemented RTL modules using Synopsis Design Compiler.

Unit	**Gate Count**	**Area (µm^2)**	**Power (µW)**
Data Minimization Unit			
- STFT	852240	133370	19886
— FFT+iFFT(256)		85614	12568
— Windowing		47756	7316
- Top-k Filter			
— Bitonic Sorter(C=16)	29816	3506	771
Randomized Response Unit			
- Encode	16453	1090	84.3
- Perturb			
— PRNG	361	63	12.48

5.7 Related Work

5.7.1 Architectural and System Support for Privacy

To ensure an individual's privacy and security many techniques such as homomorphic encryption [214], secure multi-party computing [215], and trusted excecution environments [216] have been developed. These techniques provide strong security guarantees, but when data needs to be aggregated and queried differential privacy is more applicable. Prior work shows that that differential privacy and the techniques above can be combined to provide stronger privacy guarantees [217, 218].

Ever since differential privacy was first proposed [219], it has been shown to be widely applicable in many fields. It has been used by the U.S. Census [220] and has been adapted in various applications such as IoT-based federated learning [221], privacy-preserving news recommendation system [222], and cyber physical systems [223]. Companies have also begun to use differential privacy in their data collection [188, 224, 225]

Lifestream [226] is a proposed temporal query processing engine that is optimized for physiological waveform and designed to be deployed in hospital machines acting as a central server that receives patient data. However, this work does not consider patient identity and privacy, so it would be important to privatize the data before it is queried. Opaque [217], like Lifestream, is designed to be used in hospitals that uses trusted execution environments such as Intel's SGX to ensure the confidentiality and privacy of the computations. However, the initial data must be aggregated before the computations. This assumes that the data is secure while it is being collected and when compiled. Likewise, systems like Airavat [227], PrivaApprox [228], and Microsoft's PINQ [224], which allow for differentially private querying and computations, also aggregate the data beforehand and suffer from the same issue.

Google's RAPPOR [188] is a platform that uses randomized response and bloom filters to ensure LDP when determining internet traffic patterns. Microsoft's telemetry collection technique [225] improves upon RAPPOR by guaranteeing privacy when continuously changing data is collected at regular intervals through α-rounding. However, these techniques were developed for software systems and require energy intensive computations.

5.7.2 Privacy in IoT Systems

Local differential privacy (LDP) assumes that every single data item is equally sensitive. However, for real world data, a significant amount of data that is being protected is not necessarily sensitive, consuming unnecessary resources. Based on this observation, Murakami et. al.[229] proposed an extension to LDP that allow fined-grained protection of only what the user considers as sensitive data. This makes it possible to improve utility while keeping the same level of privacy. This technique comes with caveat that the user is should be able to automatically classify sensitive data from those that are not.

Malekzadeh et. al. [230] proposed data transformations for sensitive sensor data which can be used to obfuscate the identification of sensitive activities. For these transformations, they used two types of autoencoders a Replacement AutoEncoder (RAE) that protects sensitive inferences and an Anonymizing AutoEncoder (AAE) that prevents user re-identification. Unlike the prior work, we propose a hardware-based solution that integrates directly into the sensor-module and fully configurable to explore different trade-off spaces in privacy and utility.

DP-Box [203] also attempts to provide a hardware solution for LDP in ultra low power systems. The authors show that in ultra low power systems where floating point units cannot be used, the Laplace mechanism (which relies on an RNG) does not guarantee differential privacy. To solve this they created a hardware module that uses resampling and thresholding techniques to manipulate and bound noise added by the RNG. While this module is lightweight and low power, it does not provide strong enough guarantees of LDP for time series data.

Similarly, the Privacy Protection Unit (PPU) [231, 232] is a proposed hardware unit that implements differential privacy. The PPU resides off-chip between the sensor and the processor, and provides access control and noise to the outputs of the sensors. This potentially requires ISA changes to allow the processor to interface with the PPU and assure private accesses, but this is only described at a high level and there has not been an implementation or discussion of the design challenges.

5.7.3 Signal Processing in Hardware

Garrido [233] discusses the trade offs between creating an STFT unit with several FFT units in parallel and using a windowing unit with a single FFT unit. He also proposes a feed forward STFT design that minimizes accumulation error. This design uses significantly less space than the parallel FFTs design, but is larger than the windowed design. In this chapter we use the windowed STFT design. Garrido [200] also discusses the different types of hardware FFT designs and their tradeoffs. This discussion informs our design decisions discussed in Section 5.5

5.8 Conclusion

Privacy is an increasingly critical consideration of system design, and while multiple large corporations have started to invest in privacy preserving technologies, there is still a great deal of room for innovation. Techniques that distribute the responsibility of privacy and avoid centralized points of naked aggregation are useful both because they lower the responsibility of aggregators and because they avoid single points of failure. Even if we are to trust a few entities with our most private data, there is now (and likely always will be) an appetite for our data beyond our ability to carefully examine.

We propose a new framework for architecturally supported privacy management that combines information theoretic methods with randomized response-based local different privacy to enable private aggregation of wearable time-stamped sensor data. Key to this solution is a privacy-preserving Data Minimization Unit which uses Short-Time Fourier Transform that allows mutual information reduction by using a filter for lossy reconstruction of input signals. As a demonstration, we successfully evaluate the effectiveness of our technique for sensor data such step count and BPM from real world accelerometer and electrocardiagram sensor readings. Over all we find that a carefully reduced disclosure, when coupled with random response, can unlock parts of the design space not reachable by random response alone. In several cases the error could be reduced by a factor of **3x** or more under the same privacy budget and have up to **72%** improvement in privacy for the same utility tolerance. Furthermore we find that the hardware overhead of such an implementation is quite small and we find the proposed solution does not have significant overhead in terms of chip area and power consumption.

While there is always more work to be done, we believe the contributions here take us an important step closer to understanding how privacy, utility, and overhead can trade off against one another in system that seeks to aggregate private data. Computer hardware, with its lack of malleability, is a natural place to build in non-bypassable enforcement of privacy control including the separation of levels of sensor data for internal use and other levels of data for aggregation. The privacy threat modeling privacy threat modeling and end-to-end evaluation of this work could open future research on low-power information theoretic solutions for private computations.

Chapter 6

Conclusion and Future Directions

Looking further out, the conflict between the need to share information (to provide more optimal performance) and hide information (for privacy) is becoming increasingly fundamental in all of computer science. Threats to personal data privacy are emerging as a leading concern for users. While the European GDPR and CCPA put in place privacy and data protection requirements, the onus of implementing tools to understand and embed privacy into systems falls on engineers. Computer architects must start thinking more about privacy and provide infrastructure to enable privacy at all levels of computing.

Trace wringing was the first paper to connect the problems of compression and privacy and established a new tradeoff space between utility and leakage in the context of memory address traces. By providing a way to reason quantitatively about information leakage, it opened the doors to techniques that formulate privacy as a verifiable and programmable system requirement.

6.1 Future Directions

The widespread use of public photography and the low cost of video capture has unlocked new computational and algorithmic approaches to entertainment, transportation, robotics, and many other fields, e.g. using computer vision. Unfortunately, the data-driven nature of modern approaches leaves many questions about user privacy unanswered. At the same time, due to growing reliance on custom accelerators, kernels, compilers, etc., the gap between hardware and software design is growing. The labyrinthine complexity of these designs, combined with manifestations of computer vision and machine learning, has made it increasingly challenging to reason about and implement verifiable privacy across the stack.

I believe that the design of private systems is not only possible, but through a concerted effort that bridges the gap between hardware, software, and algorithms, *we can find optimal solutions*. Through my interdisciplinary dissertation work, I have demonstrated the ability to speak to experts in a variety of fields and brought together ideas from architecture, system design, privacy, computer vision, machine learning together to produce impactful contributions. I wish to explore privacy-preserving and privacy-maximizing tools and methods for architecture and system design. Below, I outline four areas in my research program which I believe will solve pressing problems and deliver real-world impact.

Software-Hardware-Security Codesign. Recently, microarchitectural attacks have spurred universal interest in the development of secure computing. But it is crucial to understand how differing mechanisms in security, hardware, and software domains must be combined to minimize performance overheads while achieving sufficient threat model coverage. In a recent study [10], I make the case for software-hardware-security codesign. Broadly, this is a design process where feedback is used to iteratively guide the system toward performance and threat model goals. But for the various emerging secure technologies like trusted execution environments (TEEs), homomorphic encryption (HE), and differential privacy (DP), we must first expose usable abstractions for effective codesign interactions. In order to either automate or make the knowledge transfer between domains efficient, we need to provide a well-established vocabulary at the interface to collaborate effectively. Grounded in my experience of building system-level tools like Charm [8] and PyRTL [9, 5], I look forward to building tools that automate this constraint-driven iterative process and provide verifiable guarantees.

Performant Privacy. Computer architecture and systems were not developed with privacy or security in mind: performance was always the focus. Rather than treating privacy as a constraint and restricting systems in order to be private, I believe we can design *secure, private, and performant systems*, without having to give up either. Consider for example, system-level privacy methods that minimize leakage. These data minimization ideas can be likened to ideas from approximate computing. In many applications, using "approximate-private" data can lead to significant performance benefits and reduce communication overheads. In applications where data privacy and utility can be precisely defined, optimal privacy-preserving systems are within reach. Additionally, this would be the ideal means of achieving privacy in resource-constrained and edge systems.

Verifiably Private Architectures. While privacy models and methods are fast becoming a popular area of study, there is little work done at the hardware or architectural level. For example: when an application promises to leak fewer than n bits per second, how can we ensure that the application is staying within this limit? Using methods similar to trace wringing and trace scrubbing, we can begin by quantifying information leakage. Once a reasonable bit leakage threshold is set, our techniques would ensure that users' sensitive data is transferred from their personal devices and processed by cloud providers without exceeding a privacy bit budget per time unit. This measure-

and-enforce approach can be used to build verifiable, privacy-preserving systems from the bottom up. In the future, I will explore privacy *baked in to the architecture* to support confidential computing.

Private Machine Perception. Machine perception systems sense their environment, interpret and compute on sensor data, and take an action. The sensors at the heart of these systems are in close proximity to rich sources of potentially private data. Precisely defining privacy and utility early in the dataflow pipeline and restricting information flow to connected servers will significantly benefit privacy efforts. A challenging first task will be to understand what definitions of privacy and utility are most suitable to both the performance of the system and user protections. As I have shown in my work on reverse engineering user images [2], I believe data leakage minimization methods can thwart such attacks on privacy. Combined with system-level modeling, such as Charm[8], we may be able to untangle complicated relationships governing the utility and privacy of such sensor data.

Bibliography

[1] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, *Porcupine: A synthesizing compiler for vectorized homomorphic encryption*, in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 375–389, 2021.

[2] D. Dangwal, V. T. Lee, H. J. Kim, T. Shen, M. Cowan, R. Shah, C. Trippel, B. Reagen, T. Sherwood, V. Balntas, *et. al.*, *Mitigating reverse engineering attacks on local feature descriptors*, .

[3] D. Dangwal, Z. Zhang, J. R. Crandall, and T. Sherwood, *Context-aware privacy-optimizing address tracing*, in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pp. 150–162, IEEE, 2021.

[4] D. Dangwal, W. Cui, J. McMahan, and T. Sherwood, *Trace wringing for program trace privacy*, *IEEE Micro* **40** (2020), no. 3 108–115.

[5] D. Dangwal, G. Tzimpragos, and T. Sherwood, *Agile hardware development and instrumentation with pyrtl*, *IEEE Micro* **40** (2020), no. 4 76–84.

[6] W. Cui, G. Tzimpragos, Y. Tao, J. McMahan, D. Dangwal, N. Tsiskaridze, G. Michelogiannakis, D. P. Vasudevan, and T. Sherwood, *Language support for navigating architecture design in closed form*, *ACM Journal on Emerging Technologies in Computing Systems (JETC)* **16** (2019), no. 1 1–28.

[7] D. Dangwal, W. Cui, J. McMahan, and T. Sherwood, *Safer program behavior sharing through trace wringing*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1059–1072, 2019.

[8] W. Cui, Y. Ding, D. Dangwal, A. Holmes, J. McMahan, A. Javadi-Abhari, G. Tzimpragos, F. Chong, and T. Sherwood, *Charm: a language for closed-form high-level architecture modeling*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 152–165, IEEE, 2018.

[9] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, *A pythonic approach for rapid hardware prototyping and instrumentation*, in *2017*

27th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–7, IEEE, 2017.

[10] D. Dangwal, M. Cowan, A. Alaghi, V. T. Lee, B. Reagen, and C. Trippel, *Sok: Opportunities for software-hardware-security codesign for next generation secure computing*, in *Hardware and Architectural Support for Security and Privacy*, pp. 1–9. 2020.

[11] D. Mirza, D. Dangwal, and T. Sherwood, *Pyrtl in early undergraduate research*, in *Proceedings of the Workshop on Computer Architecture Education*, pp. 1–8, 2019.

[12] D. Aboye, D. Kupsh, M. Lim, J. Mai, D. Dangwal, D. Mirza, and T. Sherwood, *Pyrtlmatrix: An object-oriented hardware design pattern for prototyping ml accelerators*, in *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pp. 36–40, IEEE, 2019.

[13] J. Fowers, D. Lo, and D. Dangwal, *Deriving a concordant software neural network layer from a quantized firmware neural network layer*, Sept. 3, 2020. US Patent App. 16/290,117.

[14] K. Wuyts and W. Joosen, *Linddun privacy threat modeling: a tutorial*, *CW Reports* (2015).

[15] D. Dangwal, V. T. Lee, H. J. Kim, T. Shen, M. Cowan, R. Shah, C. Trippel, B. Reagen, T. Sherwood, V. Balntas, A. Alaghi, and E. Ilg, *Analysis and mitigations of reverse engineering attacks on local feature descriptors*, 2021.

[16] J. Redmon and A. Farhadi, *Yolov3: An incremental improvement*, *arXiv preprint arXiv:1804.02767* (2018).

[17] D. A. Osvik, A. Shamir, and E. Tromer, *Cache attacks and countermeasures: the case of aes*, in *Cryptographers' track at the RSA conference*, pp. 1–20, Springer, 2006.

[18] L. Sweeney, *Simple demographics often identify people uniquely*, *Health (San Francisco)* **671** (2000) 1–34.

[19] N. Y. Times, *A face is exposed for aol searcher no. 4417749*, 2006.

[20] A. Narayanan and V. Shmatikov, *Robust de-anonymization of large sparse datasets*, in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 111–125, IEEE, 2008.

[21] A. Joshi, L. Eeckhout, and L. John, *The return of synthetic benchmarks*, in *2008 SPEC Benchmark Workshop*, pp. 1–11, 2008.

[22] J. Weinberg and A. Snavely, *Chameleon: A framework for observing, understanding, and imitating the memory behavior of applications*, in *PARA08: Workshop on State-of-the-Art in Scientific and Parallel Computing, Trondheim, Norway*, 2008.

[23] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam, *The vpc trace-compression algorithms*, *IEEE Transactions on Computers* **54** (2005), no. 11 1329–1344.

[24] E. Elnozahy, *Address trace compression through loop detection and reduction*, in *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, pp. 214–215, ACM, 1999.

[25] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, *Analysis of branch prediction via data compression*, *ACM SIGPLAN Notices* **31** (1996), no. 9 128–137.

[26] P. Michaud, *Online compression of cache-filtered address traces*, in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 185–194, IEEE, 2009.

[27] E. Berg and E. Hagersten, *Statcache: a probabilistic approach to efficient and accurate data locality analysis*, in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pp. 20–27, IEEE, 2004.

[28] A. Sembrant, D. Black-Schaffer, and E. Hagersten, *Phase guided profiling for fast cache modeling*, in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pp. 175–185, ACM, 2012.

[29] M. Burtscher, *Tcgen 2.0: a tool to automatically generate lossless trace compressors*, *ACM SIGARCH Computer Architecture News* **34** (2006), no. 3 1–8.

[30] M. Burtscher, *Vpc3: A fast and effective trace-compression algorithm*, in *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, pp. 167–176, ACM, 2004.

[31] M. Burtscher and M. Jeeradit, *Compressing extended program traces using value predictors*, in *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pp. 159–169, IEEE, 2003.

[32] E. E. Johnson and J. Ha, *Lossless address trace compression for reducing file size and access time*, in *International Phoenix Conference on Computers and Communications, IEEE Press, Los Alamitos, CA, USA*, pp. 213–219, 1994.

[33] O. Hammami, *Taking into account access patterns irregularity when compressing address traces*, in *Southeastcon'95. Visualize the Future., Proceedings., IEEE*, pp. 74–77, IEEE, 1995.

[34] J. R. Larus, *Whole program paths*, in *ACM SIGPLAN Notices*, vol. 34, pp. 259–269, ACM, 1999.

[35] C. G. Nevill-Manning and I. H. Witten, *Linear-time, incremental hierarchy inference for compression*, in *Data Compression Conference, 1997. DCC'97. Proceedings*, pp. 3–11, IEEE, 1997.

[36] T. M. Chilimbi, *Efficient representations and abstractions for quantifying and exploiting data reference locality*, in *ACM SIGPLAN Notices*, vol. 36, pp. 191–202, ACM, 2001.

[37] X. Gao, A. Snavely, and L. Carter, *Path grammar guided trace compression and trace approximation*, in *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pp. 57–68, IEEE, 2006.

[38] L. Eeckhout, K. De Bosschere, and H. Neefs, *Performance analysis through synthetic trace generation*, in *2000 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS (Cat. No. 00EX422)*, pp. 1–6, IEEE, 2000.

[39] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, *Automatically characterizing large scale program behavior*, *ACM SIGARCH Computer Architecture News* **30** (2002), no. 5 45–57.

[40] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng, *Taming hardware event samples for fdo compilation*, in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 42–52, ACM, 2010.

[41] M. Oskin, F. T. Chong, and M. Farrens, *HLS: Combining statistical and symbolic simulation to guide microprocessor designs*, vol. 28. ACM, 2000.

[42] D. Thiebaut, J. L. Wolf, and H. S. Stone, *Synthetic traces for trace-driven simulation of cache memories*, *IEEE Transactions on computers* **41** (1992), no. 4 388–410.

[43] J. Rodriguez-Rosell, *Empirical data reference behavior in data base systems*, *Computer* **9** (1976), no. 11 9–13.

[44] D. Ferrari, *A generative model of working set dynamics*, in *ACM SIGMETRICS Performance Evaluation Review*, vol. 10, pp. 52–57, ACM, 1981.

[45] D. Ferrari, *On the foundations of artificial workload design*, vol. 12. ACM, 1984.

[46] C. M. Olschanowsky, M. M. Tikir, L. Carrington, and A. Snavely, *Psnap: accurate synthetic address streams through memory profiles*, in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 353–367, Springer, 2009.

[47] M. M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, *Pmac binary instrumentation library for powerpc/aix*, in *Workshop on Binary Instrumentation and Applications*, 2006.

[48] J. Weinberg and A. E. Snavely, *Accurate memory signatures and synthetic address traces for hpc applications*, in *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 36–45, ACM, 2008.

[49] L. Van Ertvelde and L. Eeckhout, *Dispersing proprietary applications as benchmarks through code mutation*, in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 201–210, ACM, 2008.

[50] C. Dwork, *Differential privacy: A survey of results*, in *International Conference on Theory and Applications of Models of Computation*, pp. 1–19, Springer, 2008.

[51] A. Haeberlen, B. C. Pierce, and A. Narayan, *Differential privacy under fire.*, in *USENIX Security Symposium*, 2011.

[52] V. Bindschaedler, R. Shokri, and C. A. Gunter, *Plausible deniability for privacy-preserving data synthesis*, *arXiv preprint arXiv:1708.07975* (2017).

[53] L. Sweeney, *k-anonymity: A model for protecting privacy*, *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* **10** (2002), no. 05 557–570.

[54] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam, *l-diversity: Privacy beyond k-anonymity*, in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pp. 24–24, IEEE, 2006.

[55] N. Li, T. Li, and S. Venkatasubramanian, *t-closeness: Privacy beyond k-anonymity and l-diversity*, in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 106–115, IEEE, 2007.

[56] M. Castro, M. Costa, and J.-P. Martin, *Better bug reporting with better privacy*, *ACM SIGOPS Operating Systems Review* **42** (2008), no. 2 319–328.

[57] Y. Li, J. Ren, and J. Wu, *Quantitative measurement and design of source-location privacy schemes for wireless sensor networks*, *IEEE Transactions on Parallel and Distributed Systems* **23** (2012), no. 7 1302–1311.

[58] S. McCamant and M. D. Ernst, *Quantitative information flow as network flow capacity*, in *ACM SIGPLAN Notices*, vol. 43, pp. 193–205, ACM, 2008.

[59] A. S. Dhodapkar and J. E. Smith, *Comparing program phase detection techniques*, in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 217, IEEE Computer Society, 2003.

[60] X. Shen, Y. Zhong, and C. Ding, *Locality phase prediction*, *ACM SIGPLAN Notices* **39** (2004), no. 11 165–176.

[61] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, *Discovering and exploiting program phases*, *IEEE micro* **23** (2003), no. 6 84–93.

[62] J. A. Hartigan and M. A. Wong, *Algorithm as 136: A k-means clustering algorithm*, *Journal of the Royal Statistical Society. Series C (Applied Statistics)* **28** (1979), no. 1 100–108.

[63] R. O. Duda and P. E. Hart, *Use of the hough transformation to detect lines and curves in pictures*, *Communications of the ACM* **15** (1972), no. 1 11–15.

[64] C. Galamhos, J. Matas, and J. Kittler, *Progressive probabilistic hough transform for line detection*, in *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, vol. 1, pp. 554–560, IEEE, 1999.

[65] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et. al.*, *Scikit-learn: Machine learning in python*, *Journal of machine learning research* **12** (2011), no. Oct 2825–2830.

[66] J. Y. Joshua, R. Sendag, L. Eeckhout, A. Joshi, D. J. Lilja, and L. K. John, *Evaluating benchmark subsetting approaches*, in *Workload Characterization, 2006 IEEE International Symposium on*, pp. 93–104, IEEE, 2006.

[67] M. D. Hill, *Dinero iv trace-driven uniprocessor cache simulator*, *http://www. cs. wisc. edu/˜ markhill* (1998).

[68] A. Collette, *Python and HDF5: Unlocking Scientific Data.* " O'Reilly Media, Inc.", 2013.

[69] *Tiny aes in c*, 2014.

[70] W. Cui, Y. Ding, D. Dangwal, A. Holmes, J. McMahan, A. Javadi-Abhari, G. Tzimpragos, F. Chong, and T. Sherwood, *Charm: A language for closed-form high-level architecture modeling*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 152–165, June, 2018.

[71] S. Eddy, *Hmmer user's guide*, *Department of Genetics, Washington University School of Medicine* **2** (1992), no. 1 13.

[72] P. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, in *Annual International Cryptology Conference*, pp. 388–397, Springer, 1999.

[73] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, *Rendered insecure: Gpu side channel attacks are practical*, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2139–2153, 2018.

[74] Z. H. Jiang, Y. Fei, and D. Kaeli, *A complete key recovery timing attack on a gpu*, in *2016 IEEE International symposium on high performance computer architecture (HPCA)*, pp. 394–405, IEEE, 2016.

[75] P. Luo, Y. Fei, X. Fang, A. A. Ding, M. Leeser, and D. R. Kaeli, *Power analysis attack on hardware implementation of mac-keccak on fpgas*, in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pp. 1–7, IEEE, 2014.

[76] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, *Branchscope: A new side-channel attack on directional branch predictor*, *ACM SIGPLAN Notices* **53** (2018), no. 2 693–707.

[77] W. Hua, Z. Zhang, and G. E. Suh, *Reverse engineering convolutional neural networks through side-channel information leaks*, in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[78] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood, *et. al.*, *Deepsniffer: A dnn model extraction framework based on learning architectural hints*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 385–399, 2020.

[79] P. Lifshits, R. Forte, Y. Hoshen, M. Halpern, M. Philipose, M. Tiwari, and M. Silberstein, *Power to peep-all: Inference attacks by malicious batteries on mobile devices*, *Proceedings on Privacy Enhancing Technologies* **2018** (2018), no. 4 141–158.

[80] F. Pittaluga, S. J. Koppal, S. B. Kang, and S. N. Sinha, *Revealing scenes by inverting structure from motion reconstructions*, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 145–154, 2019.

[81] D. J. Butler, J. Huang, F. Roesner, and M. Cakmak, *The privacy-utility tradeoff for remotely teleoperated robots*, in *HRI*, 2015.

[82] T. Li and N. Li, *On the tradeoff between privacy and utility in data publishing*, in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 517–526, 2009.

[83] J. A. Goguen and J. Meseguer, *Security policies and security models*, in *IEEE Symposium on Security and Privacy*, pp. 11–20, 1982.

[84] J. R. Crandall, J. Brevik, S. Ye, G. Wassermann, D. A. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong, *Putting trojans on the horns of a dilemma: Redundancy for information theft detection*, in *Transactions on Computational Science IV*, pp. 244–262. Springer, 2009.

[85] D. Devriese and F. Piessens, *Noninterference through secure multi-execution*, in *2010 IEEE Symposium on Security and Privacy*, pp. 109–124, IEEE, 2010.

[86] A. R. Yumerefendi, B. Mickle, and L. P. Cox, *Tightlip: Keeping applications from spilling the beans.*, in *NSDI*, 2007.

[87] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, *Pin: building customized program analysis tools with dynamic instrumentation*, *Acm sigplan notices* **40** (2005), no. 6 190–200.

[88] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, *Engineering record and replay for deployability: Extended technical report*, *arXiv preprint arXiv:1705.05937* (2017).

[89] D. MacKenzie, P. Eggert, and R. Stallman, *GNU Diffutils Reference Manual*. Samurai Media Limited, 2015.

[90] R. Storn and K. Price, *Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces*, *Journal of global optimization* **11** (1997), no. 4 341–359.

[91] D. J. Barrett, D. J. Barrett, R. E. Silverman, and R. Silverman, *SSH, the Secure Shell: the definitive guide.* " O'Reilly Media, Inc.", 2001.

[92] "Project gutenberg." `https://www.gutenberg.org/wiki/Main_Page`.

[93] Y. LeCun, *The mnist database of handwritten digits*, *http://yann. lecun. com/exdb/mnist/* (1998).

[94] S. Tomar, *Converting video formats with ffmpeg*, *Linux Journal* **2006** (2006), no. 146 10.

[95] M. Badr, C. Delconte, I. Edo, R. Jagtap, M. Andreozzi, and N. E. Jerger, *Mocktails: Capturing the memory behaviour of proprietary mobile architectures*, in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 460–472, IEEE, 2020.

[96] "Champsim." `https://github.com/ChampSim/ChampSim`.

[97] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, *Vigilante: End-to-end containment of internet worms*, in *ACM SIGOPS Operating Systems Review*, vol. 39, pp. 133–147, ACM, 2005.

[98] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, *Lift: A low-overhead practical information flow tracking system for detecting security attacks*, in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 135–148, IEEE, 2006.

[99] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, *Understanding data lifetime via whole system simulation*, in *USENIX Security Symposium*, pp. 321–336, 2004.

[100] A. C. Myers and B. Liskov, *Protecting privacy using the decentralized label model*, *ACM Transactions on Software Engineering and Methodology (TOSEM)* **9** (2000), no. 4 410–442.

[101] D. E. Denning, *A lattice model of secure information flow*, *Communications of the ACM* **19** (1976), no. 5 236–243.

[102] D. E. Denning and P. J. Denning, *Certification of programs for secure information flow*, *Communications of the ACM* **20** (1977), no. 7 504–513.

[103] N. Heintze and J. G. Riecke, *The slam calculus: programming with secrecy and integrity*, in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 365–377, ACM, 1998.

[104] A. C. Myers and A. C. Myers, *Jflow: Practical mostly-static information flow control*, in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 228–241, ACM, 1999.

[105] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, *Secure program execution via dynamic information flow tracking*, in *ACM Sigplan Notices*, vol. 39, pp. 85–96, ACM, 2004.

[106] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, *Rifle: An architectural framework for user-centric information-flow security*, in *37th International Symposium on Microarchitecture (MICRO-37'04)*, pp. 243–254, IEEE, 2004.

[107] J. R. Crandall and F. T. Chong, *Minos: Control data attack prevention orthogonal to memory model*, in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 221–232, IEEE Computer Society, 2004.

[108] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, *Complete information flow tracking from the gates up*, in *ACM Sigplan Notices*, vol. 44, pp. 109–120, ACM, 2009.

[109] M. Dalton, H. Kannan, and C. Kozyrakis, *Raksha: a flexible information flow architecture for software security*, *ACM SIGARCH Computer Architecture News* **35** (2007), no. 2 482–493.

[110] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, *Flexitaint: A programmable accelerator for dynamic taint propagation*, in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 173–184, IEEE, 2008.

[111] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, *Flexible hardware acceleration for instruction-grain program monitoring*, *ACM SIGARCH Computer Architecture News* **36** (2008), no. 3 377–388.

[112] J. Clause, W. Li, and A. Orso, *Dytan: a generic dynamic taint analysis framework*, in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 196–206, ACM, 2007.

[113] J. Newsome and D. X. Song, *Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.*, in *NDSS*, vol. 5, pp. 3–4, Citeseer, 2005.

[114] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, *libdft: Practical dynamic data flow tracking for commodity systems*, in *Acm Sigplan Notices*, vol. 47, pp. 121–132, ACM, 2012.

[115] A. M. Espinoza, J. Knockel, P. Comesaña-Alfaro, and J. R. Crandall, *V-dift: Vector-based dynamic information flow tracking with application to locating cryptographic keys for reverse engineering*, in *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pp. 266–271, IEEE, 2016.

[116] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, *Tainteraser: Protecting sensitive data leaks using application-level taint tracking*, *ACM SIGOPS Operating Systems Review* **45** (2011), no. 1 142–154.

[117] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, *Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 954–968, 2019.

[118] R. Panda, X. Zheng, and L. K. John, *Accurate address streams for llc and beyond (slab): A methodology to enable system exploration*, in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 87–96, IEEE, 2017.

[119] G. Balakrishnan and Y. Solihin, *West: Cloning data cache behavior using stochastic traces*, in *IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–12, IEEE, 2012.

[120] R. Panda, X. Zheng, A. Gerstlauer, and L. K. John, *Camp: Accurate modeling of core and memory locality for proxy generation of big-data applications*, in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 337–342, IEEE, 2018.

[121] Y. Wang, G. Balakrishnan, and Y. Solihin, *Metoo: Stochastic modeling of memory traffic timing behavior*, in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 457–467, IEEE, 2015.

[122] J. Matos, J. Garcia, and P. Romano, *Enhancing privacy protection in fault replication systems*, in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 336–347, IEEE, 2015.

[123] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, *Automatic input rectification*, in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 80–90, IEEE, 2012.

[124] P. Broadwell, M. Harren, and N. Sastry, *Scrash: A system for generating secure crash information.*, in *Usenix Security Symposium*, p. 19, 2003.

[125] S. Andrica and G. Candea, *Mitigating anonymity challenges in automated testing and debugging systems*, in *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*, pp. 259–264, 2013.

[126] F. Mireshghallah, M. Taram, P. Ramrakhyani, A. Jalali, D. Tullsen, and H. Esmaeilzadeh, *Shredder: Learning noise distributions to protect inference privacy*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–18, 2020.

[127] C. Cachin, I. Keidar, and A. Shraer, *Trusting the cloud*, *Acm Sigact News* **40** (2009), no. 2 81–86.

[128] E. Eilam, *Reversing: Secrets of Reverse Engineering.* John Wiley & Sons, Inc., USA, 2005.

[129] P. Weinzaepfel, H. Jégou, and P. Pérez, *Reconstructing an image from its local descriptors*, in *CVPR*, 2011.

[130] E. d'Angelo, L. Jacques, A. Alahi, and P. Vandergheynst, *From bits to images: Inversion of local binary descriptors*, *TPAMI* **36** (2013), no. 5 874–887.

[131] A. Dosovitskiy and T. Brox, *Inverting visual representations with convolutional networks*, in *CVPR*, 2016.

[132] F. Pittaluga, S. J. Koppal, S. B. Kang, and S. N. Sinha, *Revealing scenes by inverting structure from motion reconstructions*, in *CVPR*, 2019.

[133] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, and W. Joosen, *A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements*, *Requirements Engineering* **16** (2011), no. 1 3–32.

[134] A. Alahi, R. Ortiz, and P. Vandergheynst, *Freak: Fast retina keypoint*, in *CVPR*, 2012.

[135] D. G. Lowe, *Object recognition from local scale-invariant features*, in *ICCV*, 1999.

[136] Y. Tian, X. Yu, B. Fan, F. Wu, H. Heijnen, and V. Balntas, *Sosnet: Second order similarity regularization for local descriptor learning*, in *CVPR*, 2019.

[137] L. Sweeney, *k-anonymity: A model for protecting privacy*, *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* **10** (2002), no. 05 557–570.

[138] C. Salter, O. S. Saydjari, B. Schneier, and J. Wallner, *Toward a secure system engineering methodolgy*, in *Proceedings of the 1998 workshop on New security paradigms*, pp. 2–10, 1998.

[139] S. Myagmar, A. J. Lee, and W. Yurcik, *Threat modeling as a basis for security requirements*, in *Symposium on requirements engineering for information security (SREIS)*, vol. 2005, pp. 1–8, Citeseer, 2005.

[140] P. Torr, *Demystifying the threat modeling process*, *IEEE Security & Privacy* **3** (2005), no. 5 66–70.

[141] T. UcedaVelez, *Real world threat modeling using the pasta methodology*, *OWASP App Sec EU* (2012).

[142] M. Morana, "Wiley: Risk centric threat modeling: Process for attack simulation and threat analysis-tony ucedavelez, marco m. morana. accessed on 09/05/2016."

[143] P. Saitta, B. Larcom, and M. Eddington, *Trike v. 1 methodology document [draft]*, *URL: http://dymaxion. org/trike/Trike v1 Methodology Documentdraft. pdf* (2005).

[144] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, {*GAZELLE*}*: A low latency framework for secure neural network inference*, in *USENIX*, 2018.

[145] H. Durrant-Whyte and T. Bailey, *Simultaneous localization and mapping: part i*, *IEEE robotics & automation magazine* **13** (2006), no. 2 99–110.

[146] J. Miers, *Brandenburg gate*, 2008. [Online; accessed February 1, 2021].

[147] O. Ronneberger, P. Fischer, and T. Brox, *U-net: Convolutional networks for biomedical image segmentation*, in *MICCAI*, Springer, 2015.

[148] A. Radford, L. Metz, and S. Chintala, *Unsupervised representation learning with deep convolutional generative adversarial networks*, *arXiv preprint arXiv:1511.06434* (2015).

[149] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, *Imagenet: A large-scale hierarchical image database*, in *CVPR*, 2009.

[150] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, *Imagenet: A large-scale hierarchical image database*, in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[151] C. G. Harris, M. Stephens, *et. al.*, *A combined corner and edge detector.*, in *Alvey vision conference*, vol. 15, pp. 10–5244, Citeseer, 1988.

[152] Z. Li and N. Snavely, *Megadepth: Learning single-view depth prediction from internet photos*, in *CVPR*, 2018.

[153] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, *arXiv preprint arXiv:1412.6980* (2014).

[154] J. L. Schönberger and J.-M. Frahm, *Structure-from-motion revisited*, in *CVPR*, 2016.

[155] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, *Orb: An efficient alternative to sift or surf*, in *ICCV*, 2011.

[156] H. Kato and T. Harada, *Image reconstruction from bag-of-visual-words*, in *CVPR*, 2014.

[157] C. Vondrick, A. Khosla, T. Malisiewicz, and A. Torralba, *Hoggles: Visualizing object detection features*, in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1–8, 2013.

[158] Q. Zhu, M.-C. Yeh, K.-T. Cheng, and S. Avidan, *Fast human detection using a cascade of histograms of oriented gradients*, in *CVPR*, 2006.

[159] A. Mahendran and A. Vedaldi, *Understanding deep image representations by inverting them*, in *CVPR*, 2015.

[160] C. Liu, J. Yuen, and A. Torralba, *Sift flow: Dense correspondence across scenes and its applications*, *TPAMI* (2010).

[161] T. Ojala, M. Pietikainen, and T. Maenpaa, *Multiresolution gray-scale and rotation invariant texture classification with local binary patterns*, *TPAMI* (2002).

[162] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, *Communications of the ACM* (2017).

[163] P. Speciale, J. L. Schonberger, S. B. Kang, S. N. Sinha, and M. Pollefeys, *Privacy preserving image-based localization*, in *CVPR*, 2019.

[164] M. Geppert, V. Larsson, P. Speciale, J. L. Schönberger, and M. Pollefeys, *Privacy preserving structure-from-motion*, ECCV, 2020.

[165] M. Shibuya, S. Sumikura, and K. Sakurada, *Privacy preserving visual slam*, *arXiv preprint arXiv:2007.10361* (2020).

[166] M. Dusmanu, J. L. Schönberger, S. N. Sinha, and M. Pollefeys, *Privacy-preserving visual feature descriptors through adversarial affine subspace embedding*, *arXiv preprint arXiv:2006.06634* (2020).

[167] Z. Ren, Y. Jae Lee, and M. S. Ryoo, *Learning to anonymize faces for privacy preserving action detection*, in *CVPR*, 2018.

[168] T. Li and L. Lin, *Anonymousnet: Natural face de-identification with measurable privacy*, in *CVPRW*, 2019.

[169] M. S. Ryoo, B. Rothrock, C. Fleming, and H. J. Yang, *Privacy-preserving human activity recognition from extreme low resolution*, *arXiv preprint arXiv:1604.03196* (2016).

[170] N. Raval, A. Machanavajjhala, and L. P. Cox, *Protecting visual secrets using adversarial nets*, .

[171] Z. Wu, Z. Wang, Z. Wang, and H. Jin, *Towards privacy-preserving visual recognition via adversarial training: A pilot study*, in *ECCV*, 2018.

[172] F. Pittaluga, S. Koppal, and A. Chakrabarti, *Learning privacy preserving encodings through adversarial training*, in *WACV*, 2019.

[173] Z. W. Wang, V. Vineet, F. Pittaluga, S. N. Sinha, O. Cossairt, and S. Bing Kang, *Privacy-preserving action recognition using coded aperture videos*, in *CVPRW*, 2019.

[174] N. Vishwamitra, B. Knijnenburg, H. Hu, Y. P. Kelly Caine, *et. al.*, *Blur vs. block: Investigating the effectiveness of privacy-enhancing obfuscation for images*, in *CVPRW*, 2017.

[175] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft, *Privacy-preserving face recognition*, in *International symposium on privacy enhancing technologies symposium*, 2009.

[176] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, *Efficient privacy-preserving face recognition*, in *International Conference on Information Security and Cryptology*, 2009.

[177] R. Yonetani, V. Naresh Boddeti, K. M. Kitani, and Y. Sato, *Privacy-preserving visual learning using doubly permuted homomorphic encryption*, in *ICCV*, 2017.

[178] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, *Intriguing properties of neural networks*, *arXiv preprint arXiv:1312.6199* (2013).

[179] A. Nguyen, J. Yosinski, and J. Clune, *Deep neural networks are easily fooled: High confidence predictions for unrecognizable images*, in *CVPR*, 2015.

[180] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, *Evasion attacks against machine learning at test time*, in *Joint European conference on machine learning and knowledge discovery in databases*, pp. 387–402, Springer, 2013.

[181] N. Akhtar and A. Mian, *Threat of adversarial attacks on deep learning in computer vision: A survey*, *IEEE Access* **6** (2018) 14410–14430.

[182] C. Dwork, *Differential privacy*, in *International Colloquium on Automata, Languages, and Programming*, pp. 1–12, Springer, 2006.

[183] C. Dwork, A. Roth, *et. al.*, *The algorithmic foundations of differential privacy.*, *Found. Trends Theor. Comput. Sci.* **9** (2014), no. 3-4 211–407.

[184] "HealthCare.gov Sends Personal Data to Dozens of Tracking Websites.." `https://www.eff.org/deeplinks/2015/01/healthcare.gov-sends-personaldata`.

[185] "Privacy Lawsuit Targets Net Giants Over 'Zombie' Cookies.." `http://www.wired.com/2010/07/zombiecookies-lawsuit`.

[186] "University of California data breach: Sensitive information of staff, students leaked." `https://portswigger.net/daily-swig/\university-of-california-data-breach-sensitive-information\-of-staff-students-leaked`.

[187] T. Wang, J. Blocki, N. Li, and S. Jha, *Locally differentially private protocols for frequency estimation*, in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 729–745, 2017.

[188] Ú. Erlingsson, V. Pihur, and A. Korolova, *Rappor: Randomized aggregatable privacy-preserving ordinal response*, in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pp. 1054–1067, 2014.

[189] J. Tang, A. Korolova, X. Bai, X. Wang, and X. Wang, *Privacy loss in apple's implementation of differential privacy on macos 10.12*, 2017.

[190] P. M. Scholl and K. Van Laerhoven, *A feasibility study of wrist-worn accelerometer based detection of smoking habits*, in *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 886–891, IEEE, 2012.

[191] Q. Riaz, A. Vögele, B. Krüger, and A. Weber, *One small step for a man: Estimation of gender, age and height from recordings of one step by a single inertial sensor*, *Sensors* **15** (2015), no. 12 31999–32019.

[192] N. Neverova, C. Wolf, G. Lacey, L. Fridman, D. Chandra, B. Barbello, and G. Taylor, *Learning human identity from motion patterns*, *IEEE Access* **4** (2016) 1810–1820.

[193] A. Wood, M. Altman, A. Bembenek, M. Bun, M. Gaboardi, J. Honaker, K. Nissim, D. R. O'Brien, T. Steinke, and S. Vadhan, *Differential privacy: A primer for a non-technical audience*, *Vand. J. Ent. & Tech. L.* **21** (2018) 209.

[194] W. Hu, A. Ardeshiricham, and R. Kastner, *Hardware information flow tracking*, *ACM Comput. Surv.* **54** (may, 2021).

[195] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, *Register transfer level information flow tracking for provably secure hardware design*, in *Design, Automation Test in Europe Conference Exhibition*, pp. 1691–1696, 2017.

[196] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, *Complete information flow tracking from the gates up*, *SIGARCH Comput. Archit. News* **37** (mar, 2009) 109–120.

[197] D. Johnson and S. Sinanovic, *Symmetrizing the kullback-leibler distance*, *IEEE Transactions on Information Theory* (2001).

[198] J. C. Duchi, M. I. Jordan, and M. J. Wainwright, *Local privacy and statistical minimax rates*, in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pp. 429–438, IEEE, 2013.

[199] Z. Li, T. J. Oechtering, and D. Gündüz, *Privacy against a hypothesis testing adversary*, *IEEE Transactions on Information Forensics and Security* **14** (2018), no. 6 1567–1581.

[200] M. Garrido, *A survey on pipelined FFT hardware architectures*, *J. Signal Process. Syst.* (July, 2021).

[201] J. B. Allen and L. R. Rabiner, *A unified approach to short-time fourier analysis and synthesis*, *Proceedings of the IEEE* **65** (1977), no. 11 1558–1564.

[202] S. L. Warner, *Randomized response: A survey technique for eliminating evasive answer bias*, *Journal of the American Statistical Association* **60** (1965), no. 309 63–69.

[203] W.-S. Choi, M. Tomei, J. R. S. Vicarte, P. K. Hanumolu, and R. Kumar, *Guaranteeing local differential privacy on Ultra-Low-Power systems*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 561–574, June, 2018.

[204] Y. Amar, H. Haddadi, and R. Mortier, *An information-theoretic approach to time-series data privacy*, in *Proceedings of the 1st Workshop on Privacy by Design in Distributed Systems*, pp. 1–6, 2018.

[205] M. Malekzadeh, R. G. Clegg, A. Cavallaro, and H. Haddadi, *Mobile sensor data anonymization*, in *Proceedings of the international conference on internet of things design and implementation*, pp. 49–58, 2019.

[206] M. Malekzadeh, R. G. Clegg, A. Cavallaro, and H. Haddadi, *Privacy and utility preserving sensor-data transformations*, *Pervasive and Mobile Computing* **63** (2020) 101132.

[207] Y. Vaizman, K. Ellis, and G. Lanckriet, *Recognizing detailed human context in the wild from smartphones and smartwatches*, *IEEE pervasive computing* **16** (2017), no. 4 62–74.

[208] P. S. Hamilton and W. J. Tompkins, *Quantitative investigation of qrs detection rules using the mit/bih arrhythmia database*, *IEEE transactions on biomedical engineering* (1986), no. 12 1157–1165.

[209] "Chisel-STFT." `https://github.com/IA-C-Lab-Fudan/Chisel-FFT-generator/tree/STFT`.

[210] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, *ASAP7: A 7-nm finFET predictive process design kit*, *Microelectronics J.* **53** (July, 2016) 105–115.

[211] "Samsung Exynos W920 Wearable Processor." `https://news.samsung.com/global/samsung-introduces-the\-industrys-first-5nm-processor-powering-the-next-generation-\of-wearables`.

[212] C. Tan, A. Kulkarni, V. Venkataramani, M. Karunaratne, T. Mitra, and L.-S. Peh, *LOCUS: Low-Power customizable Many-Core architecture for wearables*, *ACM Trans. Embed. Comput. Syst.* **17** (Nov., 2017) 1–26.

[213] S. Sarangi and B. Baas, *Deepscaletool: A tool for the accurate estimation of technology scaling in the deep-submicron era*, in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2021.

[214] C. Gentry, *Fully homomorphic encryption using ideal lattices*, *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC 09* (2009).

[215] A. C.-C. Yao, *How to generate and exchange secrets*, *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)* (1986).

[216] M. Sabt, M. Achemlal, and A. Bouabdallah, *Trusted execution environment: What it is, and what it is not*, in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, pp. 57–64, 2015.

[217] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, *Opaque: An oblivious and encrypted distributed analytics platform*, in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, p. 283–298, 2017.

[218] V. Bindschaedler, S. Rane, A. E. Brito, V. Rao, and E. Uzun, *Achieving differential privacy in secure multiparty data aggregation protocols on star networks*, *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (2017).

[219] C. Dwork, F. Mcsherry, K. Nissim, and A. Smith, *Calibrating noise to sensitivity in private data analysis*, *Theory of Cryptography Lecture Notes in Computer Science* (2006) 265–284.

[220] J. M. Abowd, *The u.s. census bureau adopts differential privacy*, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2018).

[221] Y. Zhao, J. Zhao, M. Yang, T. Wang, N. Wang, L. Lyu, D. Niyato, and K.-Y. Lam, *Local differential Privacy-Based federated learning for internet of things*, *IEEE Internet of Things Journal* **8** (June, 2021) 8836–8853.

[222] T. Qi, F. Wu, C. Wu, Y. Huang, and X. Xie, *Privacy-Preserving news recommendation model learning*, in *Findings of the Association for Computational Linguistics: EMNLP 2020*, (Online), pp. 1423–1432, Association for Computational Linguistics, Nov., 2020.

[223] M. U. Hassan, M. H. Rehmani, and J. Chen, *Differential privacy techniques for cyber physical systems: A survey*, *IEEE Communications Surveys & Tutorials* **22** (2020), no. 1 746–789.

[224] F. Mcsherry and R. Mahajan, *Differentially-private network trace analysis*, *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM - SIGCOMM 10* (2010).

[225] B. Ding, J. Kulkarni, and S. Yekhanin, *Collecting telemetry data privately*, *Advances in Neural Information Processing Systems* **30** (2017).

[226] A. Jayarajan, K. Hau, A. Goodwin, and G. Pekhimenko, *Lifestream: A high-performance stream processing engine for periodic streams*, in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, (New York, NY, USA), p. 107–122, Association for Computing Machinery, 2021.

[227] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, *Airavat: Security and privacy for mapreduce.*, pp. 297–312, 07, 2010.

[228] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe, *PrivApprox: privacy-preserving stream analytics*, in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, (USA), pp. 659–672, USENIX Association, July, 2017.

[229] T. Murakami and Y. Kawamoto, *Utility-optimized local differential privacy mechanisms for distribution estimation*, in *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, (USA), pp. 1877–1894, USENIX Association, Aug., 2019.

[230] M. Malekzadeh, R. G. Clegg, A. Cavallaro, and H. Haddadi, *Privacy and utility preserving sensor-data transformations*, *Pervasive Mob. Comput.* **63** (Mar., 2020) 101132.

[231] M. Maycock and S. Sethumadhavan, *Hardware enforced statistical privacy*, *IEEE Computer Architecture Letters* **15** (2015), no. 1 21–24.

[232] S. Sethumadhavan, *Hardware-enforced privacy*, *Computer* **49** (2016), no. 10 10–10.

[233] M. Garrido, *The feedforward short-time fourier transform*, *IEEE Transactions on Circuits and Systems II: Express Briefs* **63** (2016), no. 9 868–872.