Building Trusted Execution Environments

Dayeol Lee

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-96
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-96.html

May 13, 2022

Building Trusted Execution Environments

by

Dayeol Lee

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Spring 2022

Building Trusted Execution Environments

Abstract

Building Trusted Execution Environments

by

Dayeol Lee

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Krste Asanović, Chair

Trusted Execution Environments (TEEs) offer hardware-based isolation, which protects the integrity and confidentiality of the in-use data of programs against various threats. Many hardware vendors have produced various TEE-enabled chips. However, there has been only a little public research on building TEEs. Building a TEE with different threat models and functionalities relies on design-space exploration. For example, a TEE must quickly adapt to various evolving threat models. In addition, a TEE can have different functionality requirements, which should not impact security guarantees. This thesis discusses research challenges in exploring the TEE design space. First, this thesis motivates why a TEE should not have a fixed threat model by demonstrating a novel off-chip side-channel attack on a TEE. Next, this thesis proposes Keystone, a software framework that enables building TEEs based on various needs, such as threat models and functionality requirements. Furthermore, this thesis discusses how to extend TEE functionality without breaking security guarantees using incremental verification.

To my family and my wife.
In loving memory of my grandmother,
Soonkeum Ko (1934-2021),
who will be very proud of me.

Contents

List of Figures

List of Tables

Acknowledgments

My journey toward a Ph.D. has been full of excitement, but also there were lots of challenges and difficulties. I would not have completed the journey successfully without immeasurable support and assistance from my colleagues, friends, and family.

First, I would like to express my sincere appreciation to my advisor, Krste Asanović. Krste allowed me to be a part of a great research group and provided his best advice that immensely helped me. He always encouraged me to do research I am passionate about. Thankful for the advice, I was able to explore many different exciting topics, including security, architecture, and systems. I am also deeply indebted to Dawn Song. Dawn was my unofficial adviser, who constantly motivated and encouraged me throughout my Ph.D. Dawn always inspired me whenever I felt demotivated or frustrated. Without her encouragement, I would not have gone through the difficulties. I would like to thank Sanjit A. Seshia and Raluca Ada Popa for providing their best guidance and helpful feedback on many research projects. Sanjit never hesitated to spend his time giving very detailed feedback, which helped me gain a lot of insights into formal methods. Raluca made me fascinated by my first security research project and provided her full support. Without her, I may not have started security research. I would like to appreciate all faculty for their profound belief in my work and unparalleled support.

I also would like to thank David Kohlbrenner, Shweta Shinde, and Chia-Che Tsai for being my academic mentors when they were at UC Berkeley. I was very fortunate to work with them and learn many things. For example, I learned about coloring diagrams for colorblindness, academic integrity, and rigorous security analysis from David; writing techniques, presentation skills, and stress management from Shweta; and engineering skills, critical thinking, and systems knowledge from Chia-Che. As those who have already gone through Ph.D., David, Shweta, and Chia-Che shared knowledge and provided valuable advice, not to mention their invaluable contributions to the papers.

It was a great pleasure to work closely with many great people. I wish to thank Kevin Cheang for collaboration throughout the writing of the dissertation. He was very dedicated to all of the projects we collaborated on and provided insightful suggestions and unwavering assistance. He also prevented my Ph.D. life from being boring. I thank Alexander Thomas, Catherine Lu, Gui Andrade, and Stephan Kaminsky for contributing to the Keystone project, Ian Fang for contributing to the Membuster project, and Pranav Gaddamadug and Cameron Rasmussen for contributing to the formal verification projects. Special thanks to Alex and Cathy for contributing so much to Keystone and helping me submit a paper during their busiest times.

I would also like to extend my gratitude to incredible people in the ADEPT, RISE, and other labs, including Alon Amid, David Biancolin, Aditya Chopra, Hasan Genc, Abraham Gonzalez, Ameer Haj Ali, Qijing Jenny Huang, Adam Izraelevitz, Sagar Karandikar, Hansung Kim, Jack Koenig, Kyle Kovacs, Seah Kim, Donggyu Kim, Kevin Laeufer, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Albert Ou, Nathan Pamberton, Arya Reais-Parsi, Colin Schmidt, Edward Wang, Lisa Wu, and Jerry Zhao. I thank all lab administrators and staff at the EECS department for creating inclusive and productive working environments.

My internships at Intel and Google have also helped me write this dissertation. I would like to thank Mona Vij, Anjo Vahldiek-Oberwagner, and Dmitrii Kuvaiski for having me at Intel Labs and providing me with challenging research projects for TEEs. Special thanks to Mona and Anjo for continuing collaboration after the internship and giving constructive advice. I thank Satnam Singh, Jade Philipoom, and Ben Blaxill for having me at Google and allowing me to participate in the SilverOak project. I could learn a lot about the formal reasoning of low-level software. I also thank my internship buddy Samuel Gruetter, who spent his valuable time teaching me Coq.

Finally, I dedicate my thesis to my family and my wife. I thank my parents, Gwanhee and Jeongsook, and my sister, Hajung, for supporting me and believing in me. My parents taught me to always live my own life and do my best, which made me who I am today. My wife, Seoyeon, has been standing by me throughout all my struggles, difficulties, and accomplishments. I love you, I am grateful to have you in my life.

Chapter 1

Introduction

1.1 Trusted Execution Environments (TEEs)

Remote computation has gained popularity along with the digitalization of everything. A massive amount of user data is generated every second, transferred to the cloud, and processed to create new values. In the last decade, the shift toward cloud computing [14, 25] has also been amplified not only by its cost efficiency, reliability, and programmability, but also by many resource-hungry applications such as machine learning and data analytics. The remote computation pandemic has raised one of the most challenging research problems: *protecting remote data during computation*.

Researchers have put an immense effort into protecting data in use. Traditionally, privileged software such as the operating system (OS) protects the in-use data by isolating virtual address spaces. However, such software-based isolation is not sufficient for remote computation for two reasons. First, the software could have been modified by someone who can physically access the remote device. For example, a cloud provider can install an arbitrary OS on their machines and provide them to customers. Second, even if software is trusted, it could have been compromised by other tenants or attackers. In either case, the data owners are unable to detect the software compromise, leading to the complete loss of control over their data. A few promising solutions are to use cryptographic technologies such as homomorphic encryption (HE) [69] or multi-party computation (MPC) [71, 223], which allow remote computation while keeping inputs and outputs encrypted. As they rely on mathematically hard problems, they offer near-perfect confidentiality and integrity of remote data. However, they are often a few orders of magnitude slower than native computation, even with state-of-the-art hardware acceleration [170].

Alternatively, *Trusted Execution Environments* (TEEs) [126, 53, 65, 13, 96, 110, 50] can fill the gap by combining hardware-based isolation mechanisms with efficient cryptographic schemes. TEEs are isolation technology that uses various hardware mechanisms, usually combined with low-level trusted software, to evict unnecessary code from the trusted computing base (TCB). In a nutshell, TEEs aim to protect code integrity, data integrity, and data confidentiality from various software adversaries, including a compromised OS. In general, TEEs provide a program with an exclusive memory region where the program can reside. TEEs use hardware mechanisms

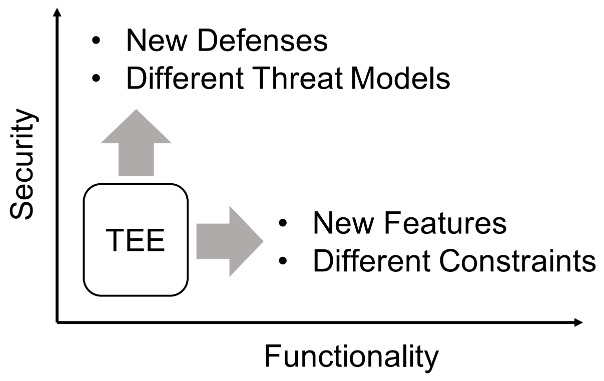

Figure 1.1: A 2-Dimensional Design Space of TEEs

to protect the memory so that even privileged software cannot arbitrarily read from or write to the memory. TEEs also arbitrate all context switches of the program, making the execution context entirely obscure to the rest of the system. Additionally, most TEEs offer remote attestation, which allows the user of the TEE to cryptographically verify that the program has been initialized and isolated. Remote attestation allows the user to securely launch remote programs and provision secrets without trusting privileged software.

All major CPU vendors have introduced TEE-equipped processors (e.g. ARM Confidential Compute Architecture [12], Intel SGX [126], and AMD SEV [6]). They differ in some design decisions, yet share the same idea: hardware-based isolation for a small TCB. Recent years have seen a wide range of efforts to migrate existing programs into those TEEs [209, 158, 160, 142, 146]. Many studies have also shown that TEEs can protect cloud services [24, 15], databases [160], big-data computations [178, 57, 33], secure banking [114], blockchain consensus protocols [116, 130, 161], smart contracts [228, 47, 26], machine learning [142, 202, 232, 156, 106], and network middleboxes [72, 73], to mention a few. The demand for confidential remote computation will continue to grow and necessitate TEEs in almost all processors in the next decade.

1.2 Challenges of Building TEEs

Although many hardware vendors have already built TEEs in their commercial products, there has been little public research on designing and building them. On the other hand, building TEEs involves rigorous security analysis and design-space exploration, which involves many research questions. How to build a trustworthy TEE without relying on a single company? What is a reasonable threat model? How to build a TEE resilient to side-channel attacks? How to reduce the cost of building TEEs? The thesis focuses on a few research challenges in the TEE design space exploration (Figure 1.1).

Threat Model. Each vendor TEE has a particular threat model that fits a specific use case. However, there is no industrial or academic consensus on what minimum adversary capabilities to consider for a TEE. Early decisions on the threat model can influence the design for a long time. For example, Intel SGX [126] insists on not defending against side-channel attacks. As a result, SGX is broken every year by side-channel attacks, including destructive ones [35, 219, 109, 212, 176, 135, 177, 175].

Customizability. Existing TEEs are delicate to customize, as they do not allow programmers to customize their TEE based on workload. It is difficult to analyze the trade-offs of different design decisions. Many workload characteristics, such as memory access pattern, working-set size, and concurrency, affect workload performance. Thus, a TEE must provide a modular and flexible design that can benefit wide-ranging workloads.

A Lack of Open Implementation. The lack of research on designing and building TEEs can be attributed to the lack of open implementation. Most vendor TEEs leverage proprietary hardware implementations, which prevents researchers from analyzing or modifying the design. Thus, many studies have relied on a few public documents published by the companies to gain only little knowledge of internal details. Therefore, exploring the design space is often limited to modifying a small part of TEE [218, 113] or software around TEE [209, 24, 184, 146]. In addition, a lack of open implementation means a lack of trust; it is precarious to fully trust a proprietary TEE implementation driven by a single company.

Secure Modification. Exploring the design space of TEEs involves not only threat models but also functionality and performance requirements. However, a patch for new functionality or performance improvement should not affect TEE security. Thus, a TEE needs to support an efficient modification and verification flow to support various workloads and platforms.

1.3 Summary of Research Contributions

The summary of contributions of this thesis are as follows:

- Chapter 2 provides the relevant background of TEEs. The chapter begins with a brief history, explains the main characteristics, and describes a few limitations.
- Chapter 3 shows that a TEE should not overlook side-channel threats, which are not present in the threat models of the major vendor TEEs to date. The chapter presents an off-chip side-channel attack on Intel SGX that is powerful enough to recover most of the sensitive data in a program. The demonstration of the attack motivates why a TEE design should not disregard side channels.
- Chapter 4 presents Keystone, which is an open framework for building TEEs. The framework is based on RISC-V instruction set architecture, enabling full transparency of software

and hardware. Keystone shows how one can separate security and functionality in a TEE design to enhance *customizability*. The chapter presents how Keystone can help explore various memory protection techniques against different threat models. Keystone is now used by hundreds of researchers for different prototypes and experiments.

- Chapter 5 proposes an efficient way to formally reason about a TEE modification by showing how TEE can support simple memory sharing. The chapter shows that *incremental verification* on a high-level specification enables an agile formal verification of a specification, which can be easily implemented by existing TEEs.
- Chapter 6 concludes the thesis by presenting the implications and potential future work.

1.4 Acknowledgment of Collaborative Work and Funding

I have authored most of the content and figures in this thesis and have adapted some parts from jointly authored publications or submissions. The detailed contributions of the collaborators follow.

Chapter 3 is based on the submission "An Off-Chip Attack on Hardware Enclaves via the Memory Bus." [109] Chia-Che Tsai wrote a fuzzy pattern-matching algorithm to extract sensitive access patterns from the trace. He also collaborated on modifying and debugging QEMU [164], Graphene [209], and SGX driver for the attack. Dongha Jung collected the trace from the experiments using the memory interposer and the signal analyzer. Ian T. Fang helped with the experiments and participated in discussions to share his brilliant ideas. Raluca Ada Popa and Chia-Che Tsai provided insightful guidance and feedback throughout the duration of the project.

Chapter 4 is based on the submission "Keystone: an Open Framework for Architecting Trusted Execution Environments." [110] David Kohlbrenner contributed to Keystone implementation, including cache partitioning, edge call interface, fixing many bugs, and improving code quality. Shweta Shinde extensively analyzed existing TEEs and compared their trade-offs. Both Shweta Shinde and David Kohlbrenner contributed to making critical design decisions for Keystone and writing the paper. Krste Asanović and Dawn Song provided valuable feedback throughout the project.

Chapter 5 is based on the submission "Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing." Kevin Cheang contributed to formalizing the TAP$_C$ model and proving the SRE property. Pranav Gaddamaduguav contributed to translating the TAP model from Boogie to UCLID5. His thesis [67] describes the contribution in detail. Alexander Thomas modified the Darkhttpd benchmark to use Cerberus. Catherine Lu wrote a program with Sqlite3 that uses Cerberus. Both Alex and Cathy have significant contributions to the implementation of Cerberus. Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A. Seshia, and Krste Asanović provided guidance and feedback.

The submissions were in part based on work supported and funded by National Science Foundation (NSF) grants CNS-1228839, CNS-1405641, CNS-1700512, NSF CISE Expeditions Award

CCF-1730628, TWC-1518899, Center for Long-Term Cybersecurity; DARPA grant N66001-15-C-4066; ADEPT Lab industrial sponsors and affiliates Intel, HP, Futurewei, Seagate, NVIDIA, and SK Hynix; and RISE Lab insdustrial sponsors and affiliates Sloan Foundation, Alibaba, Amazon Web Services, Ant Financial, ARM, Captial One, Ericsson, Facebook, Google, Microsoft, Scotiabank, Splunk, and VMware. I was partly supported and funded by the Kwanjeong Educational Foundation throughout the writing of this thesis. Any opinions, findings, conclusions, or recommendations in this thesis are solely those of the author and do not necessarily reflect the position or the policy of the sponsors.

Chapter 2

Background

2.1 The History of TEEs

TEEs first appeared in the mobile device industry, where the network operators wanted to put restrictions on mobile devices and prevent users from exploiting their networks. Now, TEEs are more popular for confidential computing in the cloud, where the users want to protect their data against cloud service providers or other companies while using cloud services. This section describes a brief history of TEEs by starting with early security solutions based on hardware.

2.1.1 Early Security Solutions Based on Hardware

The early TEEs appeared for mobile platform security in around 2000. The network operators needed restrictions and protection on end-user devices, which drove the development of hardware-aided security solutions from mobile chip vendors such as Nokia or Texas Instruments. The initial TEEs – although they were not named "TEE" back then – leveraged hardware support to protect International Mobile Equipment Identity (IMEI) or a set of parameters for radio frequency transmission in mobile devices [17, 120]. By relying on hardware that is immutable and has a narrow interface, mobile systems were able to effectively reduce the attack surface of the security functions in mobile devices.

On the server-side, commercial products such as Trusted Platform Modules (TPMs) [21, 99] or crypto co-processors like IBM 4758 [58] had also adopted the idea of leveraging trusted hardware. They were mainly supplementary pieces of hardware that provided some security functions but did not have a general computation capability. TPMs implement an attestation protocol to validate the integrity of the privileged software, which is in charge of most of the security. Thus, any generic program running on the server can rely on the privileged software and its integrity. Crypto co-processors have enabled the secure computation of specific operations such as random-number generation or public-key cryptography.

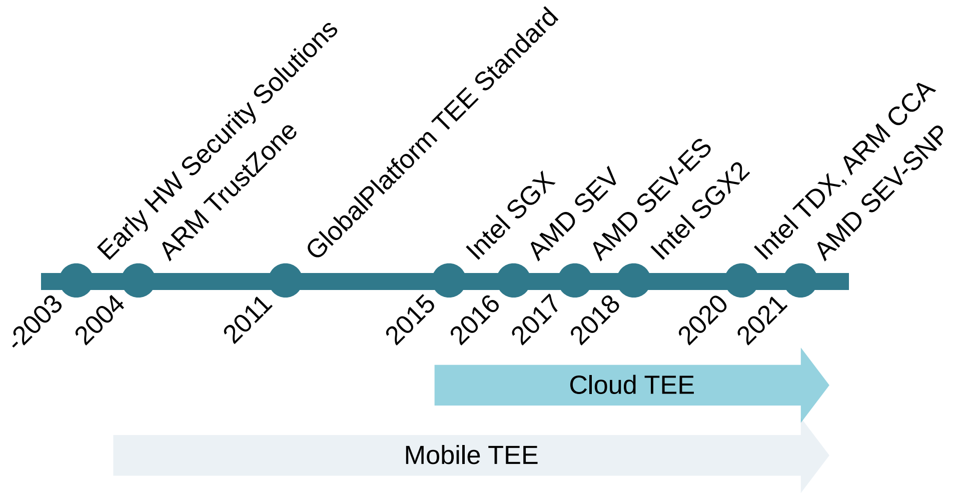

Figure 2.1: The history of Trusted Execution Environments from mobile devices to cloud computing

2.1.2 Rise of Mobile TEEs

In 2004, Arm introduced TrustZone [200], which is a system architecture that separates two parallel *execution worlds* (i.e., secure and non-secure worlds) within a single processor [200]. TrustZone isolated the secure world from the non-secure world using an additional bit in the system address bus and allowed a general program to run in the secure world's address space with access to the non-secure world. Interestingly, the first white paper of TrustZone describes the secure execution world as a *trusted execution environment*.

The term TEE had been widely used in the industry for marketing purposes but was not defined until Open Mobile Terminal Platform (OMTP) first did it in 2009 [1, 145, 207]. The OMTP-defined role of TEEs is to protect the copyrights of code and data of mobile applications. In 2011, the term was developed and standardized by GlobalPlatform [198]. GlobalPlatform standardized the TEE API and described that the TEE offers digital contents protection, authentication of security-critical code (e.g., digital payments), and integrity of system code (e.g., firmware). Along with the exponential growth of the mobile market through the mid-2010s, ARM TrustZone-based TEEs became popular in mobile devices such as smartphones and embedded devices [208, 20, 172]. In 2014, Linaro and ST-Ericsson open-sourced OP-TEE [146], a TEE based on ARM TrustZone that complies with the GlobalPlatform TEE standard.

2.1.3 Intel SGX and Advances in Academic Research

While industry-driven mobile TEEs were widespread, there was less academic work, presumably due to a lack of public resources and the limited usage of TEE [17]. In 2013, Intel revealed Secure Guard Extensions (SGX), consisting of additional instructions in Intel processors to support

TEE [126]. The architecture for SGX was substantially different from mobile TEEs. First, instead of separating secure and non-secure worlds in the entire system, SGX separates secure and non-secure address spaces within a single process. SGX referred to the secure address space as an *enclave*, which became a popular alternative terminology for SGX-like TEEs. SGX allowed mutually distrusting enclaves to run on the same system with the same security guarantees. Also, SGX targets generic trusted computing in Intel processors, which means that virtually anyone can develop any programs using their TEE.

SGX became publicly available in the sixth-generation Intel processors in 2015, and has inspired several open-source software frameworks such as Haven (2014) [24], SCONE (2016) [15], and Graphene-SGX (2017) [209] to support generic computation with SGX. Also, a large body of research on applications, vulnerabilities, functional or security extensions of SGX emerged.

2.1.4 Confidential Computing with Cloud TEEs

The last few years have seen a rise of cloud TEEs, especially for *confidential computing* in the cloud. Most vendors produce chips with various TEEs for confidential computing and cloud services. AMD introduced Secure Encrypted Virtualization (SEV) and a series of extensions including SEV-ES (Encrypted State) and SEV-SNP (Secure Nested Paging) [6, 96, 7]. SEV technologies focus on isolating a virtual machine from the rest of the system including the hypervisor installed by the cloud service provider (CSP). Intel enabled a larger SGX memory size to accommodate more cloud TEE usage. Intel has also developed Trust Domain Extensions (TDX) [89], which aims to isolate and encrypt cloud VMs sililar to SEV. The trend shows that both the major server-class CPU vendors are focusing on the cloud TEE market [1].

Building on this support, TEEs are becoming a popular option for various remote computations at scale. Many startups such as Opaque [2], Anjuna [3], and Fortanix [4] have started to build practical TEE-based software systems such as databases, data analytics, and machine learning. Confidential Computing Consortium (CCC) under the Linux Foundation [5] has initiated an industry-wide collaboration on standardizing and encouraging TEE-based confidential computing.

2.2 The Key Characteristics of TEEs

This section describes the key characteristics of TEEs. Figure 2.2 compares a TEE-based software stack with a traditional one to highlight a few promising aspects of TEEs.

[1] Intel announced that they are dropping SGX from PCs [206]

[2] https://opaque.co/

[3] https://www.anjuna.io/

[4] https://fortanix.com/

[5] https://confidentialcomputing.io/

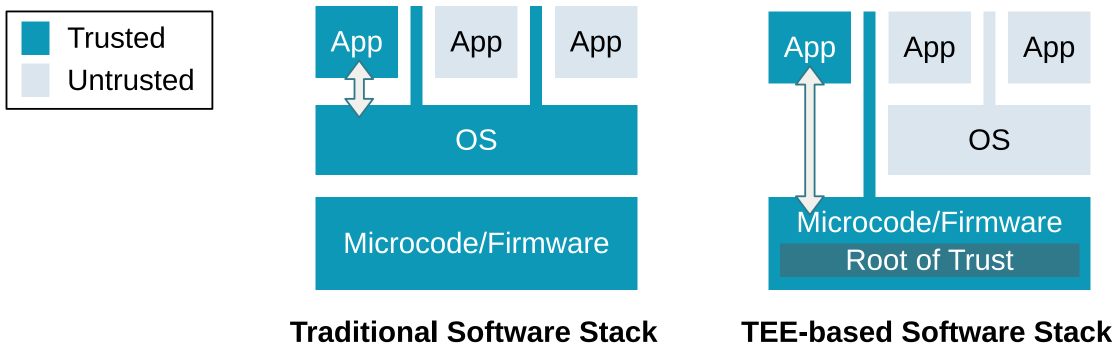

Figure 2.2: Traditional vs. TEE based software stacks

2.2.1 Hardware-aided Memory Isolation

A key idea of TEEs is to isolate a part of the physical memory by using hardware mechanisms rather than using software-managed virtual memory. Such techniques are *hardware-aided* memory isolation as they use non-traditional hardware components. For example, Intel SGX reserves Processor Reserved Memory (PRM) in the main memory, which is access-controlled and managed entirely by the microcode and a hardware extension in the memory controller. ARM TrustZone isolates the secure world from the non-secure world by adding one bit to the system bus and filtering out memory transactions. Some TEEs also perform encryption on the isolated memory contents with hardware-owned keys. The hardware-aided memory isolation techniques offer powerful protection against both software and hardware adversaries depending on the threat model.

2.2.2 Narrow Interface

In a traditional software stack, each of the user programs cannot directly interact with various hardware resources. Instead, the operating system virtualizes them and provides the user with an interface, which ends up becoming very wide and includes hundreds of system calls, device drivers, and shared memory. In contrast, a TEE can be viewed as vertical integration of software and hardware for security as the program directly interacts with various hardware-level components such as microcode or firmware via a narrow interface (Figure 2.2). A TEE platform only provides tens of functions to the user program, to which it is easier to apply strict rules to monitor and sanitize execution.

2.2.3 Minimized Trusted Computing Base

The *trusted computing base (TCB)* is the set of all components in a computer system critical to security. The size of TCB usually refers to the number of lines of code, the number of functions, or the number of external interfaces. A large TCB does not necessarily but empirically mean a large attack surface. Thus, reducing the size of TCB usually helps reduce the number of potential vulnerabilities. TEEs can reduce the size of TCB from an OS with millions of lines of code to firmware or microcode with only thousands of lines of code. Thus, a program has a smaller attack surface in TEEs than in the traditional software stack. In addition, minimizing the complexity of TCB provides the ease of formal reasoning. For example, applying automated formal reasoning to a program involves removing unbounded loops or model refinement, which are demanding with complex TCB [136, 101, 65].

2.2.4 Remote Attestation

Many cloud TEEs support *remote attestation* to provide cryptographic proof that the program inside the TEE is in a known state expected by a remote verifier. The remote attestation leverages the *chain of trust* constructed by a few cryptographic primitives. The chain of trust is backed by a *silicon root of trust*, which contains a tamper-proof secret key or identifier fused in hardware. The silicon root of trust usually measures the firmware or the boot loader and uses the key to sign the measurement. The measurement and the signature pair act as cryptographic evidence of the integrity of the firmware certified by the root of trust. In addition to the remote attestation, the root of trust often provides trusted functions such as a true random number generator (TRNG), a monotonically increasing counter, a trusted timer, or side-channel resilient crypto accelerators.

2.3 Limitations of Existing TEEs

This thesis will address the research challenges of building TEEs. However, the TEEs themselves also have some limitations related to their challenges. This section discusses the limitations of TEEs and explains how they are related to the later sections.

2.3.1 Missing Common Threat Model

Historically, TEEs have been driven mainly by the industry, with leading companies using the same term to indicate different technologies. For example, SGX, TDX, SEV, and TrustZone are all referred to as TEE, whereas their threat models are significantly different from each other. Each vendor TEE has a specific threat model that fits the product's business model. For this reason, means there is no common threat model that everyone should expect from all of the TEEs. This makes it hard for the users to assess how secure the TEE is and to what extent. Moreover, a company can change its TEE threat model even within a single line of products. For example, the early version of Intel SGX has integrity protection on the physical memory, but it was later dropped to allow larger Enclave Page Cache (EPC) sizes in the Ice Lake server generation [190]. Another example is AMD's SEV, where they gradually added protections on register states (SEV-ES) and physical memory integrity (SEV-SNP) over the course of a few years.

Different threat models have different trade-offs among security, performance, and functionality. However, it is hard to explore them with the vendor TEE, as their implementations are proprietary and assume a fixed threat model (at least within one generation of products). Chapter 4 discusses why we need a framework to explore different threat models and proposes a flexible memory-management technique that enables various memory protections modularly added to the shared codebase. The proposed Keystone framework allows the TEE programmers to analyze various trade-offs on different threat models and workloads.

2.3.2 Side-Channel Attacks

Although many recent studies unveiled destructive side-channel attacks, vendor TEEs still exclude side-channel attacks from their threat models. Some of the justifying claims are that side-channel attacks have a higher bar in practice, and the program can self-mitigate them in many cases [94, 195]. However, a series of side-channel attacks have hinted that some side channels have a lower bar and higher impact than the claims [35, 219, 212, 176, 135, 177, 175].

Chapter 3 demonstrates Membuster, an off-chip side-channel attack on Intel SGX. With physical access to the machine, the attacker can snoop on the external memory bus to observe the memory access pattern of a program and recover the secret data from the access pattern. The attack suggests that it is not ideal to completely exclude side-channel attacks from the threat model of a TEE.

2.3.3 Vulnerabilities and Patches

Unknown vulnerabilities can still compromise TEEs despite their small TCB. A TEE compromise could be more detrimental than a typical software compromise as they are likely protecting more security-critical software. Thus, patches should promptly mitigate any vulnerabilities. However, as a large portion of TEE implementation is in hardware, it is hard or even impossible to fix the vulnerabilities in a short time.

Moreover, hardware vendors usually dictate such patches. For example, both SGX and SEV implementations are in proprietary hardware components such as microcode, which only allows authenticated updates issued by the vendors. Thus, it is hard to fix the vulnerability before the TEE vendors release the patches. Also, the patches are not open-source, preventing them from being publicly reviewed or modified, thus making them less credible than how open-source software ecosystems handle vulnerabilities.

Keystone (Chapter 4) consists of high-privilege software written in common programming languages like C. As the chapter will discuss, RISC-V allows Keystone to leverage hardware-based memory protection while implementing the other features in software. Keystone is fully open source, significantly lowering the bar of the reviews and the modifications.

2.3.4 Limited Programability

Section 2.2.2 discusses how a narrow interface in TEE helps security. However, this is at the cost of programmability. A narrow interface means that only a few functions are available to a program. Thus, many rich functionalities provided by the OS via system calls are likely unavailable in TEEs.

Thus, many software projects [209, 146] implement the key functionalities via a software wrapper on top of the TEE.

Chapter 5 will discuss how such an approach can end up with a suboptimal implementation. The chapter will show a formal approach that enables an agile modification of the TEE interface to support secure and efficient memory sharing.

Chapter 3

Why Your Threat Model Might Be Wrong

This chapter shows how an attacker can break the confidentiality of a hardware enclave using *Membuster*, an off-chip side channel attack. Membuster shows that even a carefully designed TEE can be broken by a side channel, which motivates why a TEE design should consider various threat models including side-channel attacks.

3.1 Introduction

As Section 2.3.2 discusses, many side-channel attacks against TEEs have been discovered [35, 215, 32, 180, 132, 36]. Side-channel attacks leak sensitive information from enclaves via architectural or microarchitectural states. For instance, controlled-channel attacks [220] use the OS privilege to trigger page faults for memory access on different pages, to reconstruct secrets from page-granularity access patterns inside the victim program. These attacks are categorized as *on-chip side-channel attacks*, where the attacker uses adversarial or shared on-chip components to reveal memory addresses accessed by the victim (Figure 3.1).

An attacker who can *physically* access the machine can perform an *off-chip side-channel attack* that directly observes the memory addresses on the *memory bus*. The memory bus, which consists of a *data bus* and an *address bus*, delivers memory requests from a CPU to an off-chip DRAM. Although the CPU encrypts the data of an enclave, all the addresses still leave the CPU unencrypted, allowing the attacker to infer program secrets from the access patterns. Since off-the-shelf DRAM interfaces do not support address bus encryption, no existing hardware enclave can prevent physical attackers from observing the memory address bus.

Several studies have hinted at the possibility of attacks based on the memory address bus [119, 49, 85]. Costan *et al.* [49] suggest the possibility of tapping the address bus, but acknowledge that they are not aware of any successful example of the attack. Maas *et al.* [119] suggest that an attacker who can collect physical memory traces of a database server can distinguish two different SQL queries operating on the same dataset. However, no work has shown how such a side channel can be exploited to break the confidentiality of an enclave.

This chapter presents Membuster, an off-chip side-channel attack on the memory address bus.

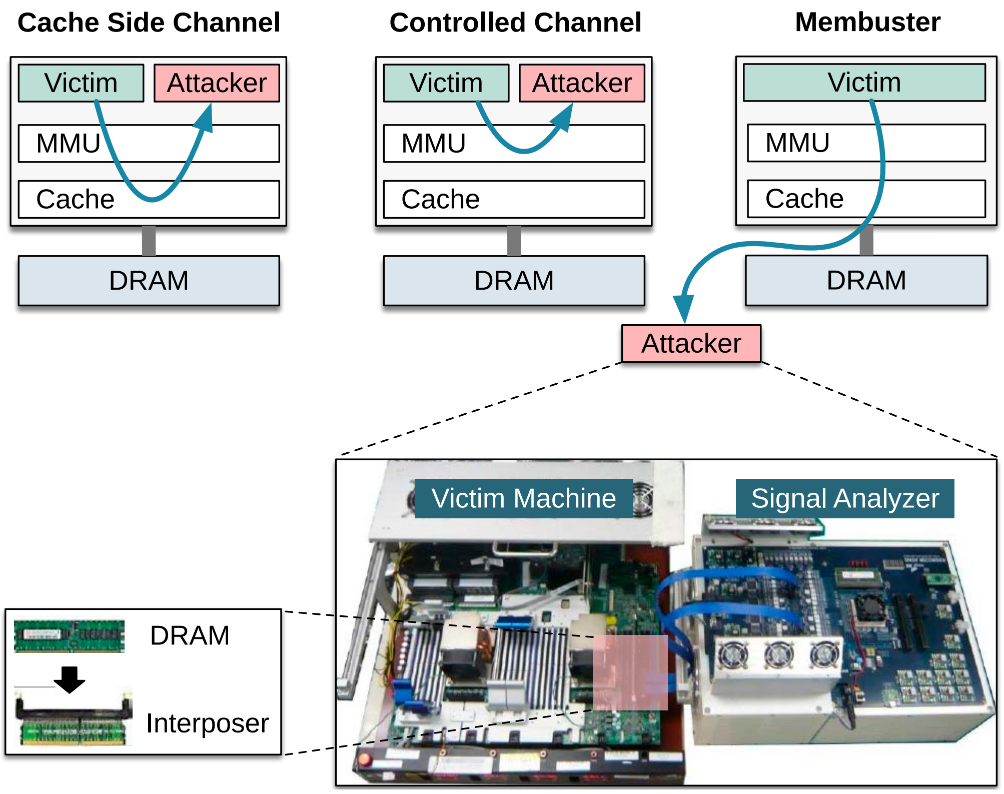

Figure 3.1: On-chip side channels compared to Membuster. The cache side-channel attack leaks addresses through a shared cache, whereas the controlled-channel attack uses adversarial memory management. Membuster leaks addresses directly through the off-chip memory bus. The photo shows an example hardware setup for the attack.

Membuster can be a substantial threat to hardware enclaves because of its unique traits compared to the existing on-chip attacks (Section 3.2.2). The need for off-chip access, despite being a disadvantage, advantages the attacker as it makes Membuster much harder to mitigate with *protected-access* solutions (Table 3.1). Recently, a wide range of tools [143, 42, 75, 185, 43] have been developed for mitigating on-chip side-channel attacks for enclaves with a reasonable overhead. These tools either partition the resources (e.g., cache) to prevent an attacker from learning information via shared resources or intercept actions (e.g., page faults) to prevent an attacker from observing the side channels. At their core, these solutions attempt to protect the memory accesses from an attacker's sight.

However, these protected-access solutions do not prevent Membuster, which observes the memory addresses off-chip and thus can bypass the protection of any on-chip solutions. To prevent Membuster on the current hardware enclave design, one must *hide* the accessed memory addresses, by making the enclave execution *oblivious* to the secret data. This requires either using oblivious algorithms [216] inside the enclave or running the enclave atop an ORAM [194, 131]. Both mechanisms bring significant performance overhead to the enclave. An alternative is to change the CPU and DRAM design to encrypt the address bus, but implementing a decryption module in DRAM can be expensive [3, 19].

The challenges to perform a robust off-chip attack are as follows:

1. **Address Translation.** The attacker needs to translate the DRAM requests into the physical addresses by reverse-engineering the mapping and to further translate them into virtual addresses of the victim enclave;

2. **Lossy Channel.** The attacker only sees DRAM requests when cache misses or write-back occurs. Since most modern CPUs have a large last-level cache (LLC), a significant portion of memory accesses do not issue any DRAM requests. Section 3.4 shows why simple methods such as *priming* the cache does not incur sufficient cache misses needed for the attack;

3. **Unusual Behaviors in SGX.** SGX has unique memory behaviors which increase the difficulty of the attack. For example, common architectural features such as disabling the cache do not work in SGX, and *paging* in SGX hides most of the memory accesses.

Section 3.3 shows how an attacker can translate the DRAM requests, and can filter out irrelevant addresses to leave only the *critical* addresses that are useful for the attack. Section 3.3.6 shows examples of applications that are susceptible to the attack. Then, Section 3.4 introduces two techniques, *critical page whitelisting* (Section 3.4.2) and *cache squeezing* (Section 3.4.4), to increase useful cache misses by thwarting page swaps and shrinking the effective cache for the critical addresses. With more cache misses, the attacker can observe more DRAM requests. These techniques do not cause detectible interference to the victim, and can be combined with cache priming to make more memory accesses visible to the attacker. The *oracle-based fuzzy matching algorithm* (Section 3.5) can create an *oracle* of the secret-to-access-pattern mapping, to identify the sensitive accesses from a sizable memory bus trace. The attacker extracts the sensitive data from the noisy memory accesses by fuzzy-matching the accesses against the oracle.

	Brasser et al. [32]	**Schwarz et al. [180]**	**CacheZoom [132]**	**FLUSH-based [36]**	**Controlled [220]**	**Membuster**
Software-Only	✓	✓	✓	✓	✓	✗
Protected-Access Fix [143, 42, 75, 185, 43]	✓	✓	✓	✓	✓	✗
Root Adversary	✓	✗	✓	✓	✓	✓
Noiseless	✗	✗	✗	✓	✓	✓
Lossless	✗	✗	✗	✓	✓	✗
Fine-Grained (64B vs. 4KB)	✓	✓	✓	✗	✗	✓
No Interference (e.g., AEX)	✓	✓	✗	✗	✗	✓
Low Overhead	✓	✓	✗	✗	✗	✓

Table 3.1: This work (Membuster) compared to previous side-channel attacks on SGX. The two boldface rows illustrate the most important distinctions. The colored cell indicates the attacker has the advantage.

The attack is demonstrated by attaching Dual In-line Memory Module (DIMM) interposer to a production system with an SGX-enabled Intel processor and a commodity DDR4 DRAM. The memory bus signals are captured to perform an off-line analysis. The attack is demonstrated by two applications, Hunspell and Memcached. Finally, the scalability of the techniques is shown by simulating the attack in modified QEMU [164].

To summarize, this chapter describes the following contributions:

- The setup of an off-chip side-channel attack on hardware enclaves and identification of the challenges for launching the attack robustly.
- Effective techniques for maximizing the side-channel information with no detectible interference nor order-of-magnitude performance overhead to the victim program.
- A fuzzy comparison algorithm for converting the address trace collected on the memory bus to program secrets.
- Demonstration and experimentation of the attack on an actual Intel SGX CPU. It is the first work that shows the practicality of the attack.

The security implications of the off-chip side-channel attacks can be pervasive because such a channel exists on almost every secure processor with untrusted memory.

3.2 Why an Off-Chip Side Channel Matters?

This section begins with the background on Intel SGX, and discusses how Membuster can be a substantial threat to hardware enclaves because of its unique traits. Table 3.1 compares Membuster with various on-chip side-channel attacks on SGX [32, 132, 180, 36, 220].

3.2.1 Intel SGX

Intel SGX has the most mature implementation and the strongest threat model against untrusted DRAM. SGX is a set of instructions for supporting hardware enclaves introduced in the Intel sixth-generation processors. SGX assumes that only the processor package is trusted; all the off-chip hardware devices, including the DRAM and peripheral devices, are considered potentially vulnerable or compromised. The threat model of SGX also includes physical attacks such as Cold-Boot Attacks [80], which can observe sensitive data from residuals inside DRAM.

An Intel CPU with SGX contains a memory encryption engine (MEE), which encrypts and authenticates the data stored in a dedicated physical memory range called the *enclave page cache* (EPC). The MEE encrypts data blocks and generates authentication tags when sending the data outside the CPU package to be stored inside the DRAM. To prevent roll-back attacks, the MEE also stores a version tree of the protected data blocks, with the top level of the tree stored inside the CPU. For Intel SGX, EPC is a limited resource; the largest EPC size currently available on an existing Intel CPU is 93.5 MB, out of 128 MB Processor's Reserved Memory (PRM). The physical pages in EPC, or EPC pages, are mapped to virtual pages in enclave linear address ranges (ELRANGEs) by the untrusted OS. If all concurrent enclaves require more virtual memory than the EPC size, the OS needs to swap the encrypted EPC pages into regular pages.

However, even with MEE, Intel SGX does not encrypt the addresses on the memory bus. As previously discussed, changing the CPU to encrypt the addresses requires implementing the encryption logic on DRAM, and thus requires new technologies such as Hybrid Memory Cube (HMC) [3, 19].

The unencrypted address bus opens up a universal threat to hardware enclaves with external encrypted memory. Komodo [65], ARM CryptoIsland [52], Sanctum [50], and Keystone [110] do not encrypt data for an external memory by default. AMD SEV [6] allows hypervisor-level memory encryption, but also does not encrypt addresses.

3.2.2 Side Channel Attacks on SGX

PRIME+PROBE

A shared cache hierarchy allows an adversary to infer memory access patterns of the victim using known techniques such as PRIME+PROBE [150, 118]. However, in PRIME+PROBE, the attacker usually cannot reliably distinguish the victim's accesses from the *noise* of other processes. The PRIME+PROBE channels are also *lossy*, as the attacker may miss some of victim's accesses while probing.

Brasser *et al.* [32] demonstrate PRIME+PROBE on Intel SGX without interfering with the enclave, but the attack requires running the victim program repeatedly to compensate for its noise and signal loss. Schwarz *et al.* [180] show that the attacker can alleviate the noise by identifying cache sets that are critical to the attack. This technique can be applied to applications that have data-dependent accesses in a small number of cache sets. CacheZoom [132] also uses PRIME+PROBE but minimizes the noise by inducing Asynchronous Exits (AEXs) every few memory accesses in the victim. This incurs a significant overhead on enclaves, and also makes the attack easily detectable [43].

Flush-based Side Channels

Other flush-based techniques such as FLUSH+RELOAD [224] and FLUSH+FLUSH [76] use a shared cache block between the attacker and the victim to create a noiseless and lossless side channel. However, these techniques cannot be directly applied to enclave memory, because an enclave does not share the memory with other processes. However, these techniques can still be used to observe the page table walk for enclave addresses [36]. Specifically, the attacker can monitor the target page tables with a tight FLUSH+RELOAD loop. As soon as the loop detects page table activities, the attacker interrupts the victim and infers page-granularity addresses. Similar to CacheZoom, this attack incurs a significant AEX overhead and thus can be detected by the victim.

Controlled Channels

Controlled-channel attacks [220] take advantage of the adversarial memory management of the untrusted OS, to capture the access patterns of an SGX-protected execution. Even though Intel SGX masks the lower 12 bits of the page fault addresses to the untrusted OS, controlled-channel attacks use sequences of virtual page numbers to differentiate memory accesses within the same page. The controlled channel is noiseless and lossless but can be detected and mitigated as it incurs a page fault for each sequence of accesses on the same page [185, 143].

3.2.3 Advantages of Membuster

As shown in Table 3.1, Membuster creates a noiseless side channel by filtering out all of the non-victim memory accesses, leaving only addresses that are useful for the attack. It can observe memory accesses with cache line granularity. Also, Membuster does not incur interference such as AEX or page fault to the victim and needs not to incur an order-of-magnitude overhead.

Several recent mechanisms, such as Varys [143], Hyperrace [42], Cloak [75], T-SGX [185], or Déjà Vu [43], have been proposed to prevent the attacker from observing memory access patterns in the victim. In general, PRIME+PROBE can be mitigated by partitioning the cache to shield the victim from on-chip attackers. This does not defeat an off-chip attacker who directly observes DRAM requests. T-SGX [185] and Déjà Vu [43] have proposed to use the Intel Transactional Synchronization Extensions (TSX) to prevent AEX or page faults from an enclave. These techniques are based on thwarting the interference (e.g., AEX, page faults) that causes the side channels [132, 36, 220]. However, Membuster does not incur such interference on enclaves, and thus cannot be thwarted through similar approaches. Thus, there is no reliable way to detect or mitigate Membuster using existing on-chip measures at the moment.

3.2.4 Related Work

Other On-Chip Attacks

Other on-chip attacks worth mentioning are speculative-based execution side channels like Foreshadow [35] or ZombieLoad [179], branch-shadowing side channels [111], denial-of-service attacks (e.g., Rowhammer [91, 213]), or rollback attacks [30, 121].

Other Off-Chip Side-Channel Attack

DRAM row buffers can be exploited as side-channels between cores or CPUs, as demonstrated in DRAMA [155]. DRAMA shows that by observing the latency of reading or writing to DRAM, the attacker can infer whether the victim has recently accessed the data stored in the same row. DRAMA shows how a software-only attacker can use DRAM row buffers as covert channels or side channels. Membuster further explores how the attacker can directly use the address bus as a side channel.

3.3 Membuster: an Off-Chip Side-Channel Attack on SGX

This section describes the basic attack model of Membuster. Further sections will refine and improve the attack. At a high level, the attacker first sets up an environment to collect the DRAM signals and waits until the victim executes some code containing data-dependent memory accesses. The attacker translates the collected signals into cache-line granularity virtual addresses.

3.3.1 Threat Model

The attack assumes the standard Intel SGX threat model in which nothing but the CPU package and the victim program is trusted. Everything else, including the OS or other applications, is untrusted and can be controlled by the attacker. External hardware devices are also untrusted, so the attacker can tap the address bus to the external DRAM. For the advanced techniques discussed in Section 3.4, the attacker may also use the root privilege to install the modified SGX driver.

To tap the memory bus, the attacker needs to have physical access to the machine where the victim is running. Such an assumption eliminates the possibility of remote attacks through either cloud environments or network connections. The attackers who may perform Membuster could be one of two types. On the server-side, these may include the employees of a cloud provider, or IT administrators of an institution, who act as insiders to leak sensitive information. On the client-side, end users may want to attack the local hardware enclaves, which protect proprietary data (e.g., licenses, digital properties, etc). The budget and knowledge requirement for the attack described

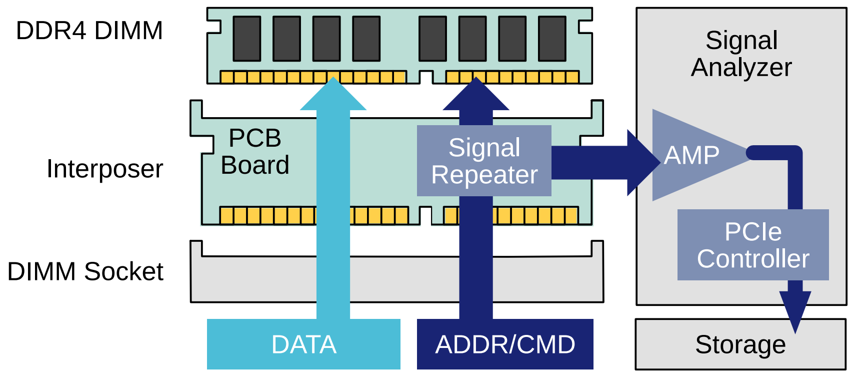

Figure 3.2: Hardware setup for a memory bus side-channel attack. DIMM interposer collects the bus signals and sends them to the signal analyzer. The attacker can use the analyzed signals to learn the memory access pattern of the victim.

in Section 3.3.2 could be an obstacle for the general public. However, the cost is manageable if the attacker has a strong motivation for obtaining the data from the enclave.

As in the controlled channel and cache side channels, Membuster assumes that the adversary has knowledge of the victim application, by either consulting the source code or reverse-engineering the application. The adversary is also aware of the runtime used by the victim application for platform support, such as the SDK libraries, library OSes, or shield systems. The experiments use Graphene-SGX [209] for platform support of the victim applications. Address Space Layout Randomization (ASLR) in the library OSes or the runtimes may complicate the extraction of secret information but generally is insufficient to conceal the access patterns completely [220]. ASLR offered by the host kernel is irrelevant because a hostile host kernel can either control or monitor the addresses where the victim enclaves are loaded.

3.3.2 Hardware Setup for the Attack

Figure 3.2 shows a detailed hardware setup for the Membuster attack. The hardware setup may vary on different CPU models and vendors. The attacker installs an *interposer* on the DIMM socket prior to system boot. The interposer is a custom printed circuit board (PCB) that can be placed between the DRAM and the socket. The interposer contains a signal repeater chip which duplicates the command bus signals and sends them to a *signal analyzer*. The analyzer amplifies the signals and then outputs the signals to a storage server through a PCIe interface.

The rest of the section will highlight the key requirements in successfully performing the attack.

Sampling Rate. The sampling rate of the interposer needs to be equal or higher than the clock rate of the DIMM in order to capture all the memory requests. A standard DDR4 clock rate ranges from 800 to 1600 MHz, while a DIMM typically supports between 1066 (DDR4-2133) and 1333 (DDR4-2666) MHz. To match with the sampling rate, the attacker can lower the DIMM clock rate if it is configurable in the BIOS.

Recording Bandwidth. The sampling rate also determines the *recording bandwidth*. For example, DDR4-2400 (1200 MHz) has a 32-bit address and a 64-bit data bus, thus the recording bandwidth for the address bus is 1200 Mbps$\times 32$ bits $= 4.47$ GiB/s. For reference, the data bus of a DDR has a $2\times$ transfer rate, as well as a $2\times$ transfer size. Hence, the bandwidth for logging all the data on DDR4-2400 will be 17.88 GiB/s.
Acquisition Time Window. The *acquisition time window* (i.e., the maximum duration for collecting the memory commands) determines the maximum length of execution that the attacker can observe. The acquisition time window equals the *acquisition depth* (i.e., the analyzer's maximum capacity of processing a series of contiguous sample) divided by the recording bandwidth of the interposer. For example, with 64 GiB acquisition depth, the analyzer can process and log the commands from DDR4-2400 up to 14 seconds.

Several vendors offer DIMM analyzers [93, 162, 163] for purchase or rental. Among them, the maximum sampling rate can reach 1200-1600 MHz, and the acquisition depth typically ranges between 4-60 GiB. One of the devices [93] can extend the acquisition time window to more than 1 hour by attaching 16 TB SSD and streaming the compressed log via PCIe at 4.8 GiB/s. Another device [163] does not disclose the memory depth but specifies that it can capture up to 1G (10^9) samples. The cost of the analyzer varies depending on the sampling rate and the acquisition depth. At the time of writing, *Kibra 480* [162] (1200 MHz, 4 GiB) costs \$6,500 per month, *MA4100* [163] (1600 MHz, 1G-samples) costs \$8,000 per month, and *JLA320A* [93] (1600 MHz, 64 GiB) costs \$170,000 for purchase.

3.3.3 Interpreting DRAM Commands

Once the attacker has finished setting up the environment, she can collect the DRAM signals at any point in time, and analyze the trace off-line. As the first step, the attacker interprets the DRAM commands collected from the interposer.

A modern DRAM contains multiple banks that are separated into bank groups. Within each bank, data (often of the same size as the cache lines) are located by rows and columns. Each bank has a row buffer (i.e., a sense amplifier) for temporarily holding the data of a specific row when the CPU needs to read or write in the row. Because only one row can be accessed in a bank at a time, the CPU needs to reload the row buffer when accessing a data block in another row.

The log collected from the DRAM interposer typically consists of the following commands:

- `ACTIVATE(Rank,Bank,BankGroup,Row)`: Activating a specific row in the row buffer for a certain rank, bank, and bank group.
- `PRECHARGE(Rank,Bank,BackGroup)`: Precharging and deactivating the row buffer for a certain rank, bank, and bank group.
- `READ(Rank,Bank,BankGroup,Col)`: Reading a data block at a specific column in the row buffer.

- `WRITE(Rank,Bank,BankGroup,Col)`: Writing a data block at a specific column in the row buffer.

Other commands such as PDX (Power Down Start), PDE (Power Down End), and AUTO (Auto-recharge) are irrelevant to the attack and thus omitted from the logs.

The DRAM commands are deconstructed into rank, bank, row, and column by simply tracing the activated row within each bank. Note that the final traces are also time-stamped by the clock counter of the analyzer. The result of the translation is a sequence of logs containing the timestamp, access type (read or write), rank, bank, row, and column in the DRAM.

3.3.4 Reverse-engineering DRAM Addressing

A physical address in the CPU does not linearly map to a DRAM address consisting of rank, bank, row, and column. Instead, the memory controller translates the address to maximize DRAM bank utilization and minimize the latency. The translation logic heavily depends on the CPU and DRAM models, and Intel does not disclose any information. Thus, the attacker needs to reverse-engineer the internal translation rule for the specific set of hardware. This has been also done by a previous study [155].

The attacker can use the traces collected from the DRAM interposer to reverse-engineer the addressing algorithm of an Intel CPU. Attacking the enclaves only needs a part of the addressing algorithm that affects the range of the enclave page cache (EPC). Then, the attacker can write a program running inside an enclave, which probes the DRAM addresses translated from the EPC addresses. The probing program allocates a heap space larger than the EPC size (93.5MB). For every cache line in the range, the program generates cache misses by repeatedly flushing the cache line and fetching it into the cache. By accessing each cache line multiple times, the attacker can differentiate the traces caused by probing from other memory accesses in the background and minimize the effect of re-ordering by the CPU's memory controller. The techniques in Section 3.3.5 are also needed for translating the probed virtual addresses to physical addresses.

Using the DRAM traces generated by probing cache lines inside the EPC, the attacker can create a direct mapping between the physical addresses and DRAM addresses (ranks, banks, bank groups, rows, and columns). The attacker can further deduce the addressing function of the target CPU (i5-8400), by observing the changing bits in the physical addresses when DRAM addresses change. As an example, the addressing function on i5-8400 is as shown in Figure 3.3. Other CPU models may implement a different addressing function, and reverse-engineering should be done for each CPU model beforehand.

3.3.5 Translating PA to VA

In order to extract the actual memory access pattern of the victim, the attacker needs to further translate the physical addresses into more meaningful virtual addresses. In general, a root-privileged attacker has multiple ways of obtaining the physical-to-virtual mappings: either by parsing the proc file `/proc/[PID]/pagemap` (assuming Linux as the OS), or using a modified

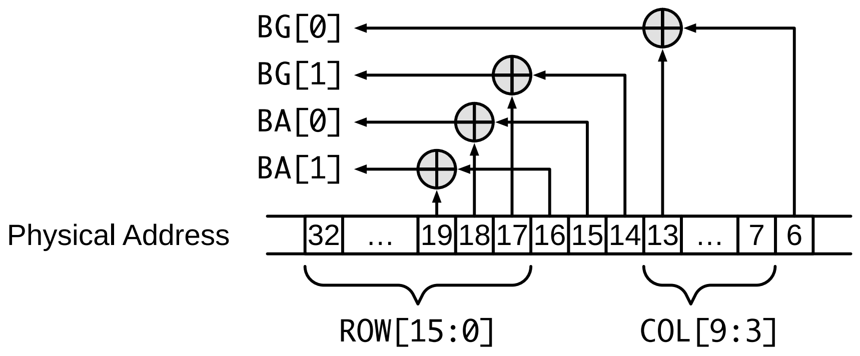

Figure 3.3: The reverse engineered addressing function of the i5-8400 CPU. The function translate a physical address to the Bank Group (BG), Bank Address (BA), Row (ROW) and Column (COL) within the DRAM.

driver. However, paging in an enclave is controlled by the SGX driver, and the vanilla driver forbids poking the physical-to-virtual mappings through the proc file system. Nevertheless, the attacker can still modify the SGX driver to retrieve the mappings.

Hence, the attacker prints the virtual-to-physical mappings in the dmesg log and ship the log together with the memory traces. During the offline analysis, the attacker uses the dmesg log as an input to the attack script. The dmesg log also contains system timings of paging, and can be further calibrated to the timestamps of the collected traces. Because paging in an enclave needs to copy the whole pages in and out of the EPC a sequential access pattern of a whole or partial page will appear in the memory traces. After calibration, the attack can successfully translate all the physical addresses to virtual addresses.

3.3.6 Membuster-Prone Application Examples

This section shows how Membuster exploits two example applications: (1) spell checking of a confidential document using *Hunspell*, and (2) email indexing cache using *Memcached*.

Hunspell

Hunspell is an open-source spell checker library widely used by LibreOffice, Chrome, Firefox and so on [86]. The controlled-channel attack [220] has shown that Hunspell is exploitable by page-granularity access patterns, which motivated us to use it as the first target of Membuster. Membuster makes the same assumptions as described in [220]; the attacker tries to infer the contents of a confidential document owned by a victim while Hunspell is spell-checking. The attacker knows the language of the document, and therefore can also obtain the same dictionary, which is publicly available.

The side-channel attacks on Hunspell are based on observing the access patterns for searching words in a hash table created from the dictionary. A simplified version of the vulnerable code is

```
// add a word to the hash table
int HashMgr::add_word(const std::string& word) {
  struct hentry* hp = (void*) malloc(sizeof(struct hentry) + word->size());
  struct hentry* dp = tableptr[i]; // Populate hp
  while (dp->next != NULL) {
    if (strcmp(hp->word, dp->word) == 0) {
      free(hp); return 0;
    }
    dp = dp->next;
  }
  dp->next = hp;
  return 0;
}
// lookup a word in the hash table
struct hentry* HashMgr::lookup(const char* word) {
  struct hentry* dp;
  if (tableptr) {
    dp = tableptr[hash(word)];
    for (; dp != NULL; dp = dp->next) {
      if (strcmp(word, dp->word) == 0) return dp;
    }
  }
  return NULL;
}
```

Figure 3.4: The Hunspell code with a data-dependent access pattern

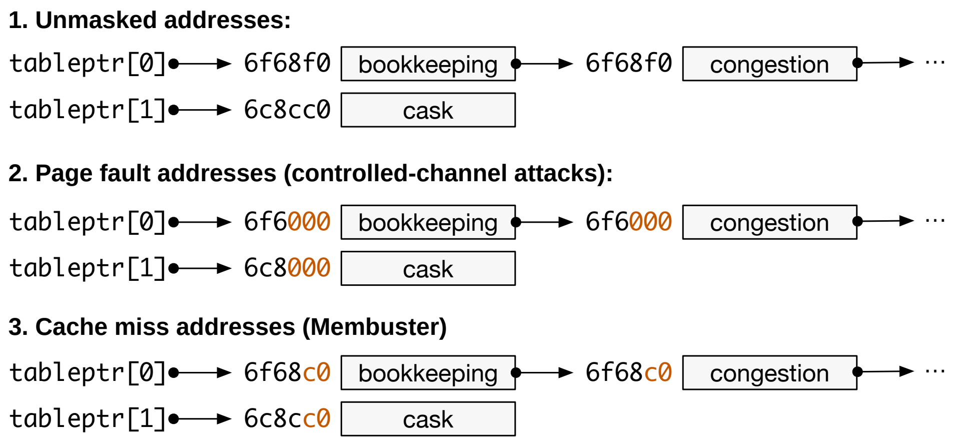

Figure 3.5: Observable address patterns in Hunspell by different attacks. Controlled-channel attacks only see page-fault addresses without the lower 12 bits, whereas Membuster can see LLC-miss addresses without the lower 6 bits.

shown in Figure 3.4. The Hunspell execution starts with reading the dictionary file and inserting the words into the hash table by calling `HashMap::add_word()`. For each word from the dictionary, `HashMap::add_word()` allocates a `hentry` node and inserts it to the end of the linked list in the corresponding hash bucket. Then, Hunspell reads the words for spell-checking and calls `HashMap::lookup()` to search the words in the hash table. Both `HashMap::add_word()` and `HashMap::lookup()` leak the hash bucket of the word currently being inserted or searched, and all the `hentry` nodes before the word is found in the linked list.

The controlled-channel attack leaks different access patterns from that Membuster observes, as the example shown in Figure 3.5. Controlled-channel attacks leak access patterns through page fault addresses, which are masked by SGX in the lower 12 bits. However, for applications like Hunspell, controlled-channel attacks can use sequences of page fault addresses to infer more fine-grained access patterns within a page. For example, although the nodes for `bookkeeping` and `booklet` are on the same page, the controlled-channel attacks can differentiate the accesses by the page addresses accessed before reading the nodes.

On the other hand, the memory bus channel can leak the addresses of each cache line being read from and written back to DRAMs, making the attacks more fine-grained than controlled-channel attacks. The attacks can differentiate the access patterns based on the addresses of each node accessed during lookups, instead of inferring through the address sequences. The granularity of memory bus attacks makes it possible to extract sensitive information even if the access patterns are partially lost due to caching.

Memcached

Memcached [66] is an in-memory key-value database, which is generally used to speed up various server applications by caching the database. Memcached is used in various services such as Facebook [139] and YouTube [128]. This example assumes that Memcached runs in an SGX enclave, as part of a larger secure system (e.g., secure mail server).

This section considers the scenario discussed by Zhang *et al.* [230], where a mail server indexes the keywords in each of the emails and the attacker can inject an arbitrary email to the victim's inbox by simply sending an email to the victim. As shown in Figure 3.6, the example assumes that the index data is stored in Memcached running in an SGX enclave. Since the attacker owns the machine, she can also perform Membuster by observing the memory bus. The attacker's goal is to use his abilities to reveal the victim's secret emails `A`, `B`, and `C`.

Memcached does not have any data-dependent control flow, but the attacker can use the memory bus side channel to infer the query sent to Memcached. Memcached stores all keys in a single hash table `primary_hashtable` defined in `assoc.c` using the Murmur3 hash of a key as an index. Each entry of the hash table is linearly indexed by the Murmur3 hash of the key. Thus Memcached will access an address within the hash table whenever it searches for a key. By observing the address, the attacker can infer the hash of the key.

Memcached dynamically allocates the hash table at the beginning of the application. The attacker can easily find out the address of the hash table by sending a malicious email to make Memcached access the hash table. For example in Figure 3.6, the attacker sends an email `D` which

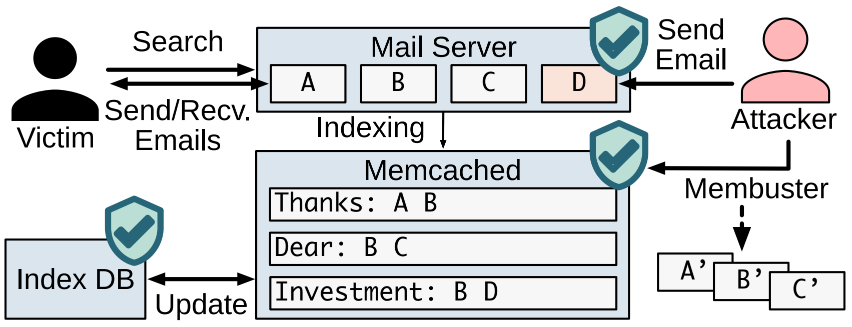

Figure 3.6: An example attack scenario where a mail server uses Memcached as an index database. `A`, `B`, `C` and `D` are the emails.

contains a word "Investment". Memcached accesses the entry, and the attacker observes the address. Since the attacker already knows the hash value of the key, she can easily find out the address of the hash table.

Next, the attacker keeps observing the memory accesses within the hash table. Once the attacker figures out the hash table address, she can reveal the hash values of the query, by observing the virtual addresses accessed by Memcached. To match the hash values with words, the attacker pre-computes some natural words and creates a hash-to-word mapping. Even though hashes can conflict, the attacker can recover most of the words by just picking a most-common word based on the statistics.

3.4 Increasing Critical Cache Misses

As previously discussed, the basic attack model of Membuster can observe memory transactions with cache-line granularity when the memory transactions cause cache misses in the last-level cache (LLC). Such an attack model is weakened in a modern processor with a large LLC ranging from 4 MB to 64 MB, causing only a small fraction of memory transactions to be observable on the DRAM bus.

This section introduces techniques to increase cache misses of the target enclaves. In a realistic scenario, an attacker only cares about increasing the cache misses within the virtual address range which leaks the side-channel information. Take the attack on Hunspell for example, the attacker only needs to observe the access on the nodes which store the dictionary words. A memory address is *critical* if the address is useful for the attack. The goal is to increase the cache misses on critical addresses, to improve the success rate of the Membuster attack.

3.4.1 Can We Disable Caching?

A simple solution to increase cache misses is to disable caching in the processor. On x86, entire cacheability can be disabled by enabling the CD bit and disabling the NW bit in the control register

CR0 ([90], Section 11.5.3). Some architectures allow disabling caching for a specific address range, primarily for serving uncacheable DMA requests or memory-mapped I/Os. For instance, on x86, users can use the Memory Type Range Register (MTRR) to change the cacheability of a physical memory range. Newer Intel processors also support page attribute table (PAT) to manage page cacheability with the attribute field in page table entries.

However, besides disabling the entire cacheability, neither MTRR or PAT can overwrite the cacheability of SGX's processor-reserved memory (PRM) [183]. The cacheability of PRM is specifically controlled by a special register called Processor-Reserved Memory Range Register (PRMRR), which can be only written by BIOS during booting. Since there is no proprietary BIOS that allows the user to modify PRMRR, the attacker effectively has no way to change the cacheability of the encrypted memory. However, since the BIOS is untrusted in the threat model of SGX, in theory, one can reverse-engineer the existing BIOS or build a custom BIOS to overwrite PRMRR. Membuster does not rely on such techniques because disabling cacheability will incur significant slowdown, making the attack easy to detect by the victim.

3.4.2 Critical Page Whitelisting

After paging (swapping), memory access in the swapped pages becomes unobservable to the attacker. Such a phenomenon is common for SGX since SGX has to rely on the OS to swap pages in and out of the EPC. Both swap-in and swap-out causes the page to be loaded into the cache hierarchy (LLC, L2, and L1-D caches), because the SGX instructions for swap-in and swap-out, i.e., `eldu` and `ewb`, require re-encrypting the page from/to a regular physical page [183]. After the instructions, the cache lines stay in the cache hierarchy until being evicted by other memory access. Currently, an Intel CPU with SGX only has up to 93.5MB in the EPC, making paging the primary obstacle to observing critical transactions on the memory bus.

On the other hand, paging also complicates the virtual-to-physical address translation, as the mappings can change midst execution. Certain observed access patterns can be used to identify the paging events. However, these patterns can also become unobservable if the page is recently swapped and most of the cache lines are still in the LLC.

Therefore, to eliminate the side effect of paging, the modified SGX driver pins the EPC pages for the critical address range. The attacker starts by identifying the critical address range of each target program. Take the Hunspell program for example. The critical memory transactions come from accessing the dictionary nodes, which are allocated through `malloc()`. For simplicity, Address Space Layout Randomization (ASLR) is disabled inside the enclave (controlled by the library OS [209]). ASLR can be defeated by identifying contiguous memory access pattern in the traces. Next, the attacker calculates the number of EPC pages needed for pinning the critical pages. For a Hunspell execution using an en_US dictionary, the total `malloc()` range is 5,604 KB. Finally, the critical address range is fed into the modified SGX driver. When the driver allocates an EPC page, it checks if the virtual address is in the critical address range and use an in-kernel flag to indicate if the page has to be pinned. The driver will never swap out a pinned page.

3.4.3 Priming the Cache

The attacker can incur cache misses by actively contaminating the cache by accessing contentious addresses. This technique is called *cache priming*, which is used in the PRIME+PROBE attack [118]. Previous work has established priming techniques for either same-core or cross-core scenarios. Some priming techniques are restricted by CPU models, especially since many recent CPU models have employed designs or features that raise the bar for cache-based side-channel attacks. However, recent studies also show that, even with these defenses, attackers continue to find attack surfaces within the CPU micro-architectures, such as priming the cache directory in a non-inclusive cache [222].

Membuster only uses cross-core priming since same-core priming requires interrupting the enclaves using AEX or page faults. The usage of cache priming in Membuster is distinctly different from existing cache-based side-channel attacks since Membuster does not require resetting the state of the cache or synchronizing with the victim. The goal of cache priming in Membuster is to simply evict the critical addresses from the cache to increase the cache misses. Also, with cache squeezing, the attacker only has to prime the cache sets dedicated to the critical addresses. These differences make it easy to apply multiple priming attacks simultaneously, as long as they all eventually contribute to increasing cache misses.

Cross-Core Cache Priming Multiple priming processes run on other cores to evict the critical cache lines from the LLC. These processes will repeatedly access the cache sets that are shared with the critical addresses of the victim. The attacker will start by identifying the critical addresses and the cache sets to prime. Then, the attacker starts the priming processes before the victim enclave, to actively evict the cache lines during execution. Take the Hunspell attack for example. Since its critical addresses are spread over all cache sets, the attacker needs to repeatedly prime all cache sets. No synchronization is required between the attack processes and the victim.

A potential hurdle for cross-core priming is to obtain sufficient memory bandwidth to evict the critical cache lines. Based on the experiments, a priming process that sequentially accesses the LLC has around 100–200MB/s memory bandwidth. Priming a 9MB LLC with 2,048 sets requires about 100 milliseconds, which is too slow to evict the critical cache lines before the lines are accessed by the victim again. For instance, Hunspell accesses a word every 2 thousand DRAM cycles (< 1 microseconds), and Memcached accesses a word every 5 million DRAM cycles (< 2.5 milliseconds). Section 3.4 will discuss, however, how an attacker can evict all the critical cache lines within a few milliseconds by pinpointing the priming process to target only 64–128 sets.

Page-Fault Cache Priming Potentially, an attacker can prime the LLC, L2, and L2-D caches on the same core with the victim, by interrupting the victim periodically. To do so, the attacker can take a similar approach to the Controlled-Channel Attack: The attacker identifies two code pages containing code around the critical memory accesses, and then alternatively protects the pages to trigger page faults. To increase cache misses, the attacker needs not to prime the cache at every page fault, but rather can prime at a low frequency. However, such a page-fault priming technique still causes a lot of interference and overhead to the victim, making it easy to detect [132] or to

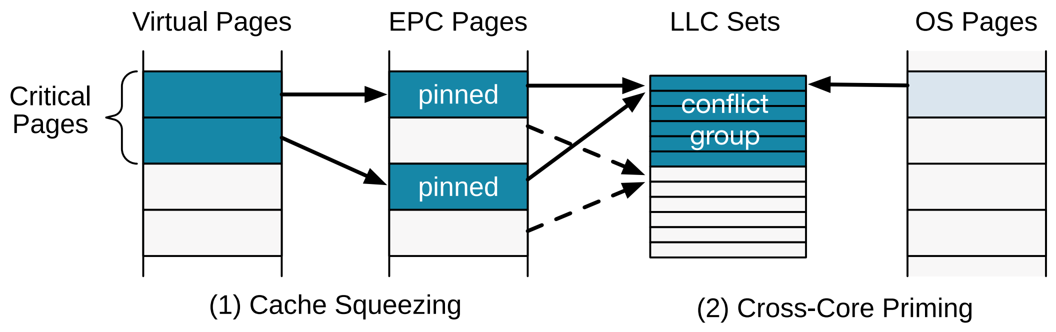

Figure 3.7: Techniques used to increase the cache miss rate with minimal performance overhead.

mitigate [185, 43]. For example, priming the cache on every 10-20 page faults incurs about $3\times$ overhead to the victim. In addition, known countermeasures, such as T-SGX [185], can effectively prevent page faults using transactional instructions. Therefore, Membuster does not use this technique.

3.4.4 Shrinking the Effective Cache Size with Cache Squeezing

As previously discussed, cache priming alone cannot create sufficient memory access bandwidth for evicting the critical cache lines in time. Therefore, Membuster leverages a novel technique called *cache squeezing*, which shrinks the effective cache size to incur more cache misses for a specific address range. The technique can be combined with non-intrusive techniques like cross-core cache priming to make Membuster a more powerful side channel.

Cache Squeezing

As the name suggests, cache squeezing can shrink the effective cache size for a given set of critical pages. By squeezing the cache that an enclave can use, the attacker can incur both *conflict misses* and *capacity misses* on LLC, therefore becoming able to observe more cache misses on the bus.

In modern processors, the L2 cache and LLC are physically-indexed. The lowest 6 bits of the physical address are omitted, given that each cache line is 64 bytes. The next s lower bits are taken as the *set index*. Each set then consists of W ways to store multiple cache lines of the same set index. For an enclave, an OS-level attacker can control the physical pages that are mapped to the enclave's virtual pages. This allows the attacker to manipulate the physical frame number (PFN) of each virtual address of the enclave, and subsequently, the higher $s-(12-6)=s-6$ bits of the set index.

Figure 3.7(1) shows how cache squeezing works in combination with page pinning. The attacker first defines the critical addresses of the victim, then maps these pages to EPC pages that share the minimum amount of cache sets. This technique requires cache pinning so that these pages will never be swapped out from the EPC. Since the OS only controls the higher $s-6$ bits of the set indices, the smallest group of physical pages that will evict each other share exactly $2^6 = 64$ sets. Such a group of physical pages is a *conflict group*. Since the maximum size of EPC is 93.5 MB, the entire cache can be partitioned to 2^{s-6} conflict groups where each conflict group can accommodate 93.5 MB/4 KB/2^{s-6} EPC pages. In the experiment, $s = 11$ (2048 sets) and $W = 12$, so each conflict group can accommodate at most 748 pages (2,992 KB). The critical address range of Hunspell, for example, is the whole `malloc()` space, which is 5,604 KB and thus requires two conflict groups. Finally, the attacker gives the critical address range to a modified SGX driver, which will only map physical pages from the selected conflict groups to any critical virtual address.

Using cache squeezing to increase cache misses has many benefits. First of all, it does not require interrupting the victim enclaves, nor does it need to incur more memory accesses in the background. All memory accesses used to push cache lines out of the L2 cache and LLC are legitimate accesses from the victim enclaves. Therefore, cache partitioning cannot defeat cache squeezing because there is no cross-context cache sharing. In fact, way-partitioning features such as Intel CAT [138] can be exploited to further shrink the effective cache sizes in combination with cache squeezing.

Cross-Core Priming with Cache Squeezing

As Section 3.4.3 mentioned, cross-core cache priming may not have sufficient bandwidth to evict the critical cache lines in time. However, cache squeezing makes the priming more effective by shrinking the effective cache size. Instead of priming all the cache sets, the attacker now only has to prime the sets of the targeted conflict groups containing the critical addresses (Figure 3.7(2)). Each group of 64 cache sets contains $W \times 4$KB, allowing the priming process to evict the part of cache within a millisecond. The priming process can run in parallel and does not affect the victim execution except causing cache contention.

Limitation

Although cache squeezing can increase the cache misses among critical addresses, it could be less effective if the victim has only a few critical addresses or a small memory footprint. If the critical addresses can only fill a small part of a conflict group ($W \times 4$ KB), the victim enclave may not be able to cause enough cache misses to benefit the attacker. For example, Memcached only has 2 MB (500 pages) of the critical address range. To fill all of the 748 pages, the top 248 frequently-accessed pages (in addition to the critical addresses) are identified through simulation, and are assigned to the same conflict group.

Note that the LLC of a modern CPU usually has a *cache slice* feature that distributes the addresses across multiple cache banks using an undocumented, model-specific mapping function. Reverse-engineering the slicing function of the target CPU is useful for further reducing the effective cache space for an enclave if the enclave has a smaller memory footprint. Reverse-engineering of slicing functions is already explored by prior papers [222], so this thesis will not discuss this technique.

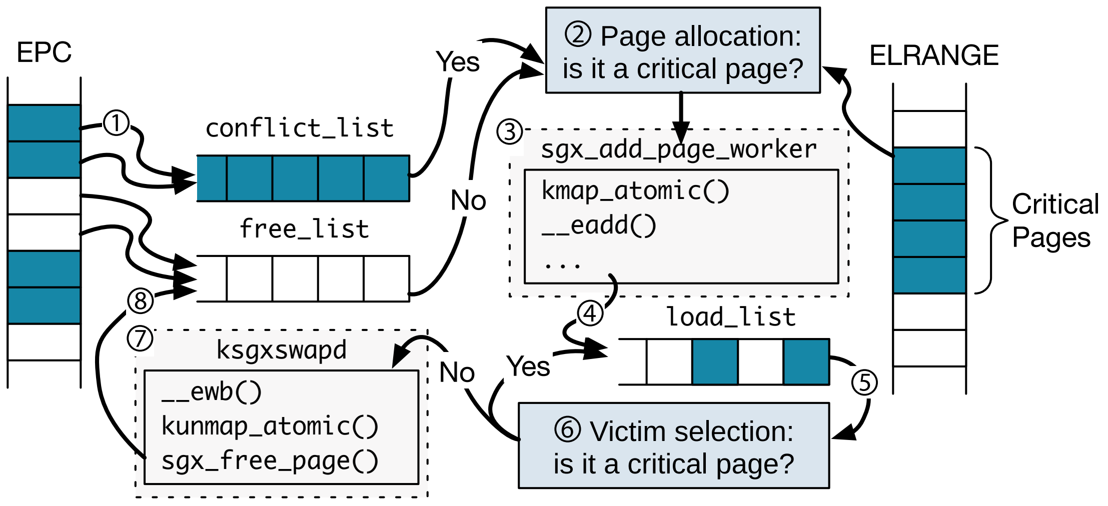

Figure 3.8: Implementation of critical page whitelisting and cache squeezing in a modified SGX driver. To ensure no swapping in the sensitive memory range, EPC pages are set aside in a separate queue. The attackers can further select the EPC pages based on set indexes or other logistics.

One can detect the cache squeezing by testing if critical addresses are mapped in an adversarial way. Since the enclave is not aware of physical address mappings by itself, it needs to experimentally detect such mapping by accessing the addresses and measure latency. However, it is challenging because (1) the victim needs to know the critical address range to detect the mapping, and (2) the enclave cannot tell if the mapping was accidental or intentional.

Implementation

Figure 3.8 shows how a modified SGX driver implements both critical page whitelisting and cache squeezing. The driver accepts parameters to specify a sensitive range within the victim application, and calculates how many conflict groups are required for the attack. ① When the driver initializes, it inserts conflicting EPC pages to a separate queue (i.e., `conflict_list`). ② When adding enclave pages, the driver checks if the virtual page number is in the critical address range. ③ The driver maps the critical pages to pages popped from `conflict_list`. ④ All of the mapped pages are added to the list of loaded pages (`load_list`). ⑤ When the driver needs to evict an EPC page, it searches the victim from the list of loaded pages. ⑥ If the selected page is a critical page, it searches again. ⑦ Only non-critical pages are evicted and the enclave continues to run. Other enclaves are not affected by the modification and can function normal with marginal overheads.

The change to the SGX driver contains only 290 lines. The SGX driver uses the `fault` operation in `vm_operations_struct` to handle EPC paging. A customized `fault` function checks the faulting virtual addresses of the enclave and then applies different paging strategies to critical and non-critical addresses. The range of critical addresses for each application was hard-coded, requiring switching the drivers for a different target. Potentially, the driver can export an API to the attackers for specifying the critical addresses. The driver also only supports one single victim enclave at a time. However, the attacker can extend the driver to target multiple enclaves simultaneously as long as the total memory usage can fit into the EPC (required for pinning).

3.5 Extracting Sensitive Access Patterns

OS techniques including critical page whitelisting, cache squeezing, and cross-core priming effectively increase the cache misses on the cache misses on critical addresses. However, the traces collected from the memory bus are still full of noise and contain no marker for splitting the critical memory accesses into iterations. Unlike controlled-channel attacks, Membuster cannot rely on repeated code addresses (e.g., from a loop) to mark and then split the critical accesses because these code addresses tend to be accessed too frequently to be evicted by the techniques. Therefore, the attacker needs to deeply analyze the memory traces offline to distill the sensitive information.

To extract the sensitive access patterns, the attacker uses four techniques for filtering the critical memory addresses and matching with a known oracle for the target application: (1) offline simulation; (2) searching the beginning of sensitive accesses; (3) fuzzy pattern matching, and (4) exploiting cache prefetching. The following sections will explain how to analyze the access pattern by using the examples from Section 3.3.6.

3.5.1 Offline Simulation

Side-channel attacks often require attackers to have some knowledge about the behaviors of the victim. For example, the controlled-channel attack on Hunspell requires the attacker to extract the virtual page addresses of the linked list nodes of each dictionary word, during an online *training phase* while attacking the victim. However, Membuster cannot perform online training with the victim as the analysis of the memory traces is performed offline. Instead, the attacker needs to generate an oracle of the victim behavior, using offline simulation of the target application.

The attacker can use a deterministic oracle for each application, given that users have adopted some publicly available data (e.g., the `en_US` dictionary). For example, during the simulation, A modified Hunspell runs in an enclave, which prints out the indexes and the addresses of linked list nodes visited for each word. Then, the output is reused as an oracle for analyzing any traces based on the same `en_US` dictionary.

As discussed earlier, ASLR in the enclaves does not invalidate an oracle, since ASLR can be easily defeated by observing the specific patterns related to binary loading. The addresses in the oracle can simply be shifted by a certain offset to be usable again.

3.5.2 Searching Sensitive Accesses

Finding the first sensitive access is critical for deciding where to start matching access patterns. Note that not all accesses to the critical addresses are sensitive. For Hunspell, allocating nodes for each word emits a long sequence of monotonically increasing virtual addresses that can be used to identify the sensitive addresses. The attacker matches the virtual addresses to the oracle, to find the *longest increasing subsequence (LIS)* of addresses as accessed in the dictionary order. After finding the LIS, the next critical access is the beginning of the sensitive addresses.

3.5.3 Fuzzy Pattern Matching

In Membuster, a part of memory addresses in a sensitive access pattern is likely to be missing due to caching. Even with cache squeezing and cross-core priming, it is almost impossible to force page misses on every critical memory access. Therefore, to analyze lossy traces, the attacker uses fuzzy pattern matching to flexibly match the traces with only parts of access patterns. As long as at least one or a few accesses of a pattern cause LLC misses, the attacker can identify the pattern as a possible result for recovery.

In fuzzy pattern matching, one address may be parsed as different access patterns of the victim for two reasons. First, within a data structure such as a linked list or a tree, the same address (an inner node) may be accessed while traversing or searching other nodes. Second, a cache line may contain multiple nodes and thus can be accessed when visiting one of the nodes. For either of the reasons, a single memory trace may be accounted for multiple possible access patterns in the oracle.

The attacker uses a simple strategy to select the best interpretation for a set of memory traces. The attacker assigns a score to each possibility based on how *complete* the traces have matched with an access pattern in the oracle. For the addresses of a tree or a linked list, the attacker assigns lower scores to the root and the first few nodes and assign higher scores to nodes that are closer to leaves or the end of the list. By collecting the top-ranking interpretations of the memory traces, an attacker can generate a list of the most probable options of the target secret. Potentially, a grammar checker or any semantic-based heuristic can help to validate or to rank the recovery results.

3.5.4 Exploiting Cache Prefetching

Finally, the cache prefetching features of CPUs can help increase the accuracy of the attack. For example, a recent Intel CPU includes *Next-line Prefetcher* and *128-byte Spatial Prefetcher*. The Next-line Prefetcher, belonging to the L2 cache, will preload the cache line next to the one that is currently accessed. The 128-bit Spatial Prefetcher, which also belongs to the L2 cache, prefetches the pairing cache line that completes the accessed cache line to a 128-byte aligned chunk into the LLC. Both prefetchers increase the number of memory accesses relevant to the secret data. Therefore, the attacker expands the range of pattern matching based on their knowledge of cache prefetching, including extending the addresses representing each secret by 64 bytes, both backward and forward. As a result, even if the CPU has cached a line, the prefetched lines may still cause cache misses and be observed on the memory bus.

Other cache prefetchers such as *Stream Prefetcher* can monitor an ascending or descending sequence of addresses from the L1 or L2 cache and can prefetch up to 20 cache lines ahead of the loaded address. Such a prefetcher generally will not improve the accuracy of the pattern matching.

However, these prefetchers can cause space pressure to caches, making cache squeezing more effective.

3.6 Attack Results

This section presents the evaluation results of the Membuster attack, based on the two vulnerable applications described in Section 3.3.6. The evaluation mainly answers the following questions regarding the Membuster attacks:

- How accurate can Membuster extract the secrets from applications that are vulnerable to such an attack?
- How do the attack techniques of Membuster impact the attack accuracy?
- How much slowdown (or interference) the various techniques will incur on the applications?
- What is the limitation of Membuster?
- How sensitive are the attack results of Membuster to the last-level cache (LLC) size of the target CPU?

This section evaluates Membuster in various settings: (1) the basic attack without any techniques (None); (2) the optimized attack with cache squeezing (SQ); (3) the optimized attack with cache squeezing combined with cross-core cache priming (SQ+PR).

3.6.1 Experiment Setup

This section describes the experimental setup of the Membuster attack. The evaluation uses both physical and simulated experiments to evaluate the effectiveness of Membuster.

Physical Experiment

Hardware Setup. Table 3.2 shows the hardware setup for the experiment. The experiment uses a machine equipped with an Intel SGX CPU. The DIMM connects to a signal analyzer via a DIMM interposer. BIOS was configured to slightly increase the DRAM supply voltage to offset the voltage drop caused by the interposer. The bus frequency is set to 1066 MHz, so the bandwidth of the analyzer is 3.97 GiB/s. With a 64 GiB acquisition depth, the attacker can log the memory bus for up to $\sim$ 16 seconds. All of the experiments have finished in a few seconds, and thus the acquisition depth is sufficient for logging all the memory requests. To achieve a wider time window, the attacker can choose an analyzer which can filter the requests by addresses [163], or which has a higher acquisition depth [93].
Victim Setup. The victim machine is running Ubuntu 16.04 and Linux kernel 4.4. The unmodified victim applications run inside enclaves via Graphene-SGX [209]. The victim may also

CPU	
Model	Intel i5-8400 (Coffee lake)
LLC Size	9 MB
LLC # Slice	6 Slices
LLC # Associativity	12-way set associative
LLC # Sets	2048
Memory	
DIMM Type	DDR4-2400 UDIMM (Non-ECC)
Capacity	8 GB
Channel/Rank/Bank/Row	1/1/16/65536
Page Size	8 KB (1 KB/package)
Max Bus Frequency	1200 MHz

Table 3.2: Hardware specification for the experiment

choose other frameworks [15] or port the applications with the SDK [191], but the choices of the frameworks do not eliminate the patterns since they do not change the program logic of the victim.
Sample Size. SK Hynix collaborated to borrow its proprietary analyzer for the experiments. Due to the limited access to the device, the attack was conducted only *once* for each setting. The experiments ran successfully despite the small sample size because the results match well with the expectations learned from the simulation.

Microarchitectural Simulation

A software simulator was used to simulate the attack prior to an actual attack because the hardware setup requires costly devices. This allows the attacker to get preliminary results. The results are then cross-validated with the results from the actual hardware setup, to verify the functional correctness of the simulation. QEMU [164], a machine emulator, was used to trace all the physical memory accesses of the guest. To capture cache misses, a modified QEMU emits all the memory requests to a cache simulator integrated from Spike [193]. The cache simulation does not implement any cycle-accurate hardware model as well as cache slicing and pseudo-LRU replacement. However, the simulation was sufficiently faithful for developing the attack scripts to analyze the real memory traces.

Enclave Simulation

The experiment simulates an enclave environment without memory encryption, using a modified Graphene-SGX library OS and a dummy SGX driver. Simulating Intel's Memory Encryption Engine (MEE) unnecessary because MEE does not affect the memory addresses accessed within the EPC. MEE generates additional access patterns for the integrity tree or EPC metadata, both of which are stored in the Processor Reserved Memory outside the EPC. The attack does not rely on any access pattern outside the EPC.

The modified Graphene-SGX library OS and the dummy SGX driver primarily simulate the transition in and out of the enclave and the paging of enclave memory, to generate similar memory access patterns as observed on the memory bus. For simulating enclave entry and exit, the modified user-tier SGX instructions (i.e., `EENTER` and `EEXIT`) in the Graphene-SGX runtime directly jump to addresses that are originally given as the enclave entry. The experiment also simulates the AEX.

To simulate EPC paging, the modified SGX driver replaces the system-tier SGX instructions, including the `ELDU` and `EWB` instructions, which swap and re-encrypt pages in and out of the EPC. These two instructions are replaced with memory copy without encryption. The memory traces from the real enclaves and from the simulation match, confirming that the results are identical.

Applications: Hunspell

The experiment uses Hunspell v1.6.2 to evaluate the effectiveness of the Membuster attack. Hunspell uses a standard `en_US` dictionary [192] with two document samples: a random non-repetitive document with 10,000 words (**Random**), and a natural-language document *Wizard of Oz* with 39,342 words (**Wizard**). For simplicity, the samples are normalized based on `en_US` dictionary, by converting non-existing words in the samples to the closet words in the dictionary. Membuster does not recover words that are reported as misspelt by Hunspell. In addition, the experiment disabled affix detection in Hunspell.

The experiment uses the pattern matching algorithm described in Section 3.5 to recover the target document from the DRAM traces collected from the Hunspell program running inside the enclave. The hardware prefetching was enabled by configuring the BIOS. To verify the result, the experiment selects an interpretation of the DRAM traces that is closet to the target document, from a set of highest-ranking results generated from the algorithm.

Application: Memcached

The experiment runs Memcached v1.5.12 as another target of the Membuster attack. In this attack, the secrets are the data being looked up in the Memcached cache. Enron email dataset [61] was used as a realistic workload for Memcached. First, the experiment computes the 4-byte hash of each word that appears in emails in the sent mail directory of each user. In total, there are about 7000 unique word entries in the dataset, which include articles and propositions. During the *training phase*, assuming the attacker is monitoring a Memcached server, the attacker can determine both the hash table address and the hash value of each word using the traces of a few queries. Then, during the *attack phase*, the attacker monitors the memory bus traffic of an enclave-protected Memcached server receiving caching requests from an trusted email server. The email server parses emails from a test data set that contains randomly selected emails with around 1000 words in total. As the Memcached server processes the caching requests from the email server, the attacker can extract the words in the emails using the Membuster attack.

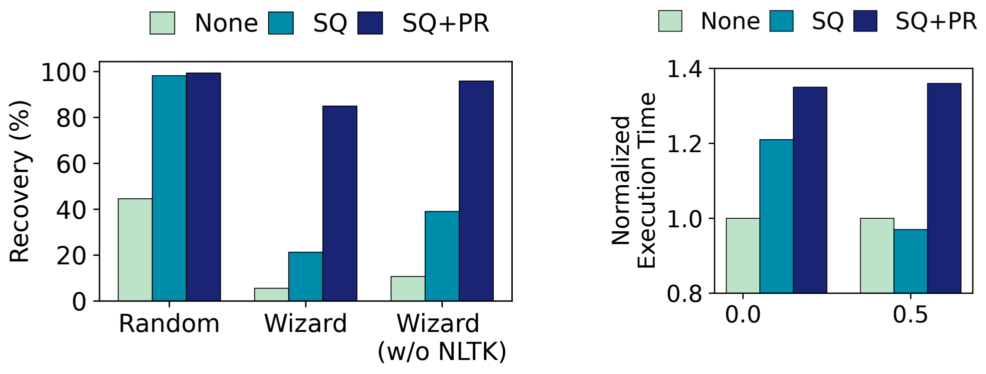

Figure 3.9: Hunspell document recovery rate (left) and normalized execution time (right) on two documents: Random document (Random) and Wizard of Oz (Wizard). The comparison is between without any techniques (None); with cache squeezing(SQ); and with cache squeezing and cross-core priming (SQ+PR). The Wizard of Oz results also show the recovery rate of uncommon words only (w/o NLTK).

Technique	Attack Accuracy	Normalized Exec. Time
None	34.1%	1.00×
SQ	82.1%	0.92×

Table 3.3: Membuster results for attacking Memcached on an SGX machine

3.6.2 Effectiveness of the Attack

Data Recovery Accuracy

Figure 3.9 (left) and Table 3.3 show the accuracy of Membuster for recovering the victim's data. The accuracy is the number of words recovered from the collected traces, compared to the number of words in the original samples. The recovery rate is higher in a non-repetitive (Random) or high-interval access pattern (Memcached) than in a repetitive access pattern (Wizard). Even without any techniques (None), Memcached and Random show 34% and 44% recovery rates, respectively. Cache squeezing recovers 96% of the random document and 82% of the Memcached query.

However, for Wizard of Oz, None or SQ can only achieve up to 21% recovery rate. The main reason is that the document contains many repetitive words, including common words such as "you" and "the" and uncommon words such as "Oz" and "scarecrow". The memory accesses for these words are likely to be cached in the LLC cache without emitting any DRAM requests. On average, each unique word in Wizard of Oz repeats 15.5 times. Without cache squeezing and cross-core priming, the attack recovers about 0.3 occurrences of each word on average. Even with cache squeezing, the attack only recovers about 2.6 occurrences.

Since cache squeezing shrinks the effective cache size for the critical addresses, cross-core priming becomes more efficient by only priming the sets of the critical addresses. Combining cache squeezing and cross-core priming (SQ+PR) achieves 85% recovery accuracy on Wizard of Oz.

Furthermore, the attacker is most likely to need only the *uncommon* words to be recovered. Common words are defined by *stopword*s from the NLTK dataset [140] which includes 179 common words (e.g., "the"). Excluding the common words allows Membuster to recover Wizard of Oz up to 95% (Figure 3.9 Wizard w/o NLTK).

Overhead and Interference

Membuster does not incur an orders-of-magnitude overhead that can be distinguishable by the victim. Figure 3.9 (right) shows the normalized execution time with different attack techniques with respect to the baseline. In general, both cache squeezing and cross-core priming have a low performance impact on the victim program, since these techniques do not interrupt the victim program. For Hunspell, cache squeezing causes up to 21% overhead to the victim, and up to 36% if combined with cross-core priming. The overheads are mainly caused by the increase of cache misses inside the victim program.

Table 3.3 also shows the end-to-end execution time of Memcached for processing the whole test set. Similar to Hunspell, the basic attack incurs no overhead on Memcached. Interestingly, cache squeezing reduces the execution time by 8% for Memcached. On a physical machine, critical page whitelisting consistently reduces the average LLC miss rate (2.9% vs. 3.6%) as well as the page fault rate. Because the physical pages of Memcached's hash table are pinned inside the enclave, and thus never get swapped out from the EPC. Thereby, within the hash table, there is no expensive paging and context switching cost that generally plagues enclave execution.

Scalability on # of Ways

The attack on the simulation environment shows the scalability of Membuster. The number of sets are fixed to $s = 2048$ that most Intel CPUs choose to have. Since the simulator does not simulate the LLC slices, the experiment increases the size of the cache by increasing the number of ways, W. To clarify, increasing the number of ways does not reflect the actual behavior of LLC with multiple slices. Even if the LLC has multiple slices, each cache line will compete with W other cache lines. Thus, increasing W makes the attack much harder, by reducing the chance of eviction of critical addresses. Note that a typical W value is between 4 and 16.

As shown in Figure 3.10, cache squeezing makes cross-core priming much more effective in general by reducing the effective cache size. cache squeezing was more scalable on Hunspell than Memcached, because Hunspell has a larger critical address range. With $W = 64$, Membuster recovered up to 83% of the random document in Hunspell and 88% of the emails in Memcached when both cache squeezing and cross-core priming have been used. Even assuming an unrealistic number of ways $W = 256$, which results in 32 MB of LLC, the attack accuracy was 77% and 40% respectively.

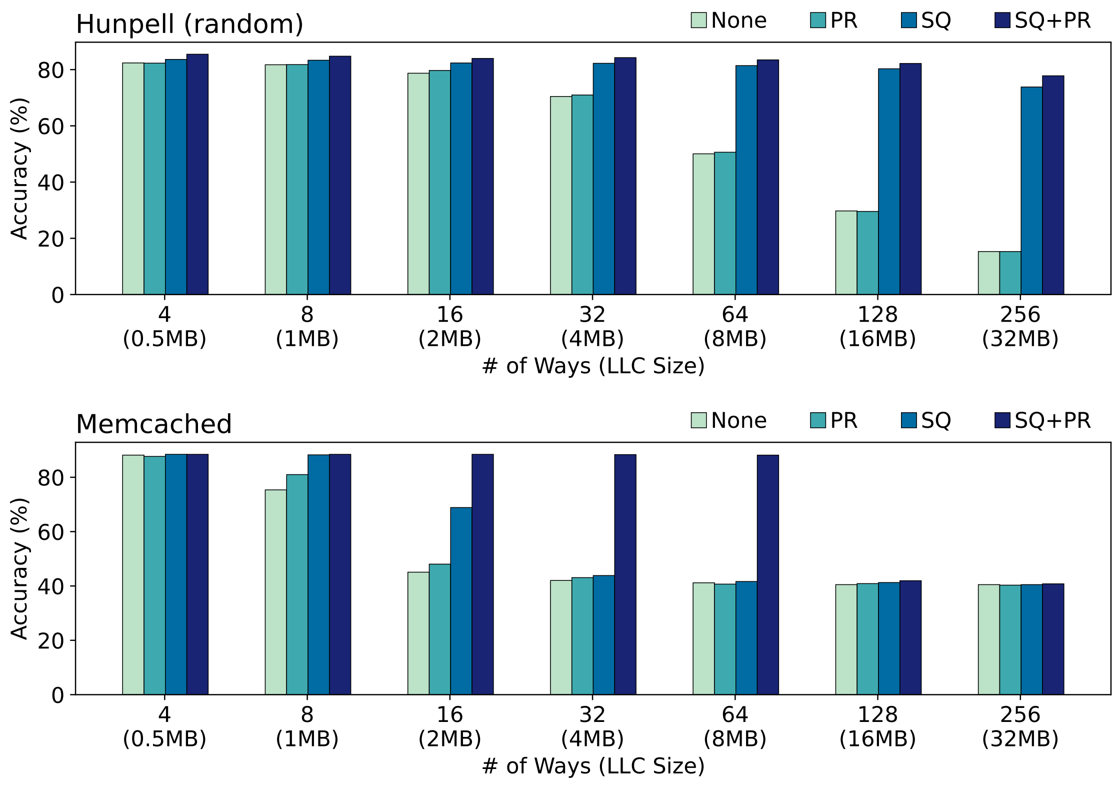

Figure 3.10: Simulation results of the attack on Hunspell (top) and Memcached (bottom).

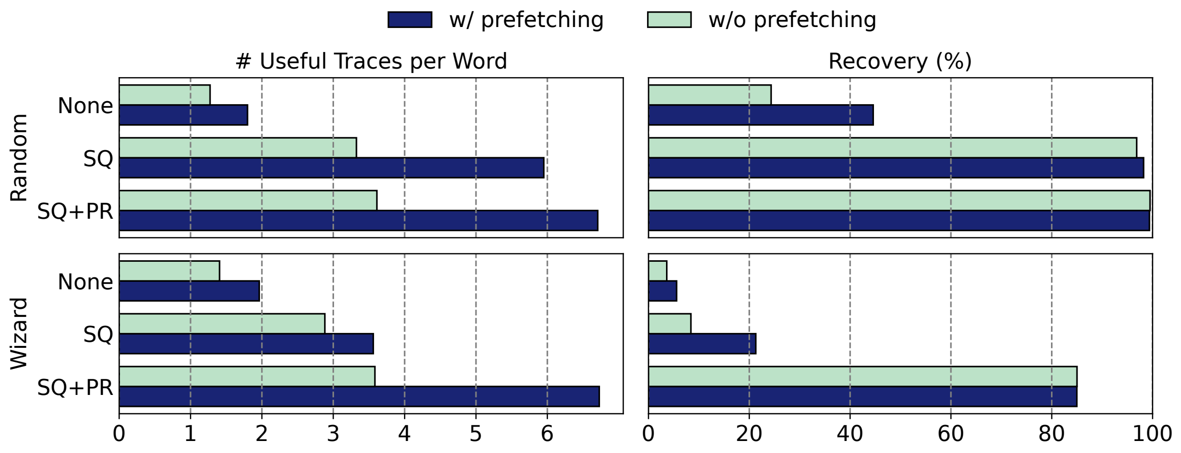

Figure 3.11: The number of useful traces per word and the document recovery rate for each experiment (with or without the hardware prefetcher).

3.6.3 Per-Application Detailed Analysis

Hunspell: Advantage of Cache Prefetching

Exploiting cache prefetching advantages Membuster. For Hunspell, the attacker recovers each word based on multiple memory accesses. If the attacker observes more traces relevant to each word, recovering the word becomes easier. Hence, if the attacker knows the presence of cache prefetchers in advance, she can use the information to correlate the prefetched addresses with each word (Section 3.5).

As shown in Figure 3.11, cache prefetching increases the average number of useful traces per word. Including prefetched addresses increases the recovery rate especially when there are very few useful traces (None and SQ). Although the improvement is marginal in the experiment, the attacker can potentially use the additional memory requests made by the cache prefetchers to extract more information from the victim.

Memcached: Advantage of Fine-Grained Addresses

To show the advantage of observing fine-grained addresses, the experiment simulated the controlled-channel attack on Memcached example. First, the entire memory trace is obtained from Memcached without simulating the cache. Then the the lower 12-bits of all addresses are masked assuming each page is 4 KB. The post-processing enabled simulating the memory trace that the controlled-channel attacker will observe. The experiment also reconstructs the attacker's hash table such that each page-granularity address maps to multiple entries in the hash table. If the attacker sees an address, she simply chooses the most common word among the possible entries.

The simulated controlled-channel attack achieves only 29% accuracy, and the recovered document was uninterpretable as it only contained common words such as "the" and "of". This shows that Membuster leverages fine-grained addresses by providing more side-channel information than coarse-grained addresses.

3.7 Implications and Limitations

Potentially, Membuster can be used in two scenarios: (1) a malicious user attacking an end device to retrieve secret data from a local enclave; (2) a malicious cloud provider or employee attacking a cloud machine to retrieve secret data from the tenants. The existence of Membuster shows the importance of physical security to enclaves just on par with software security. Ideally, in a secure cloud, one may want to separate the person who has physical access to the machine from the person who has administrative privileges. This may be achieved by a secure boot system that prevents people who have physical access from overwriting system privileges.

Limitations. Membuster leaks only memory access patterns at LLC misses. Thus, Membuster cannot observe repeated accesses to the same address within a short period. For instance, the former RSA implementation of GnuPG [70] is known to leak a private key through code addresses in the ElGamal algorithm [224]. This type of attack relies on *data-dependent branches*, as the attacker detects different code paths executed inside the victim to infer the secret. However, these vulnerabilities are difficult to exploit by Membuster, due to these code addresses being frequently executed and thus cached in the CPU. Even cache priming techniques cannot efficiently evict the code addresses in time to help the attacker retrieve the secret with high accuracy but keep the performance impact low.

In general, Membuster is more suitable for leaking *data-dependent memory loads* over a large heap or array. For instance, both the attacks on Hunspell and Memcached rely on the access patterns within a large hash table and/or linked-list objects. If the victim program only has data-dependent memory access patterns within a small region, or if the memory access is not evenly distributed, the accuracy of Membuster is likely to worsen. Besides, if the application only leaks a secret through *stores* that are dependent on the secret, Membuster may not observe the memory requests immediately. The reason is that the CPU tends to delay *write-back* of dirty data until the cache lines are evicted, making the timing of the memory requests appearing on the memory bus unpredictable.

Timing Information. Although not explored in the chapter, an attacker may exploit the timing information to attack the victim. The DRAM analyzer logs a precise timestamp for each memory request based on counting its clock cycles. Potentially, an attacker can measure the time difference between two memory traces, to infer the execution time of operation in the victim as a way of timing attacks.

Traffic Analysis. Potentially, the memory bus traffic recorded by the DRAM analyzer can be used for traffic analysis if the victim is vulnerable to this type of attacks. For instance, the attacker may analyze either the density or the volume of requests on a specific address to infer the activity or secret of the application. A complete mitigation of the attack should eliminate the timing information and has a constant traffic flow on the memory bus [3].

Multiple DIMMs or Multi-Socket. The current attack does not explore the possibility of having multiple DIMMs or multiple CPU sockets (currently not supported by SGX). However, potentially, the attacker can attach multiple DIMM interposers, and then correlate the DRAM traces using timestamps or common patterns.

Memory Controllers. A memory controller arbitrates all transactions to main memory such that it maximizes the throughput while minimizing latencies. One of the key features that may make Membuster more challenging is *transaction scheduling* where the *arbiter* reorders the transaction requests to maximize the performance. In other words, the order of the memory transactions observed by the attacker may differ from the actual order of memory accesses.

The arbitration of the memory controller does not stop an enclave from leaking sensitive access patterns. First, even if transactions are reordered, the critical addresses will still eventually appear on the memory bus. Also, the memory controller only reorders transactions within a very small time window (e.g., tens of bus cycles), which is not enough to obfuscate the critical memory accesses that occur at least every hundreds of instructions.

Generalization. Intel SGX is not the only platform affected by Membuster. Other existing platforms of hardware enclaves [65, 50, 110, 52] also do not encrypt the addresses on the memory bus. Thus, these platforms are also vulnerable to Membuster as long as the CPU stores encrypted data in external memory (e.g., DRAM). The attacker can also use the same techniques such as cache squeezing to induce cache misses on other platforms. For example, Komodo [65] allows the OS to affect the virtual address mapping, which enables the attacker to use cache squeezing. Keystone [110] measures the initial virtual address mapping for attestation, thus cache squeezing cannot be applied. However, it provides cache partitioning which can reduce the effective cache size of the enclave.

Mitigations. There are several ways to mitigate Membuster, but they are generally expensive. Oblivious RAM (ORAM) [194, 174] can make the applications execute in an oblivious manner so that the attacker cannot infer secret data based on the memory access pattern. The high performance overhead of ORAM makes it less attractive for applications that have strong performance requirements. Alternatively, additional hardware can encrypt the address bus as proposed by InvisiMem [3] and ObfusMem [19]. However, adding such a feature to commodity DRAM would be very expensive; take the cost of techniques such as Hybrid Memory Cube (HMC) [154] for an example. In-package memory such as high bandwidth memory (HBM) may relieve the needs for protection against untrusted DRAM [104], but remains an expensive alternative for production.

3.8 Summary

This chapter introduced Membuster, which is a non-interference, fine-grained, stealthy physical side-channel attack on hardware enclaves based on snooping the address lines of the memory bus off-chip. The key idea is to exploit OS privileges to induce cache misses with minimal performance overhead. This chapter also demystified the physical bus-based side channel by reverse-engineering the internals of several hardware components. Then it developed an algorithm that can retrieve application secrets from memory bus traces. This chapter demonstrated the attack on an actual SGX machine; the attack achieved similar accuracy with much lower overhead than previous attacks such as controlled-channel attacks. The attack technique is prevalent beyond Intel SGX and can apply to other secure processors or enclave platforms, which do not protect memory buses.

Chapter 4

Keystone: An Open Framework for Building TEEs

This chapter describes Keystone, an open framework for building TEEs. Keystone uses simple abstractions provided by the hardware such as memory isolation and a programmable layer underneath untrusted components (e.g., OS). By using reusable TEE core primitives, Keystone allows platform-specific modifications and flexible feature choices.

4.1 Introduction

Each vendor TEE enables only a small portion of the possible design space across threat models, hardware requirements, resource management, porting effort, and feature compatibility. When a cloud provider or software developer chooses a target hardware platform they are locked into the respective TEE design limitations regardless of their actual application needs. Constraints breed creativity, giving rise to significant research effort in working around these limits. For example, Intel SGXv1 [126] requires statically sized enclaves, lacks secure I/O and syscall support, and is vulnerable to significant side-channels [49]. Thus, to execute arbitrary applications, the systems built on SGXv1 have inflated the Trusted Computing Base (TCB) and are forced to implement complex workarounds [24, 15, 209]. As only Intel can make changes to the inherent design trade-offs in SGX, users had to wait for changes like dynamic resizing of enclave virtual memory in SGXv2 [125]. Unsurprisingly, these and other similar restriction have led to a proliferation of new TEEs on other ISAs (e.g., OpenSPARC [39], RISC-V [50, 171]). However, each such redesign requires considerable effort and only provides another fixed design point.

The hardware should provide security *primitives* instead of point-wise solutions. Thus, this chapter leverages RISC-V's primitives to construct highly customizable TEEs. One can draw an analogy with the move from traditional networking solutions to Software Defined Networking (SDN), where exposing the packet forwarding primitives to the software has led to far more novel designs and research. Such a paradigm shift in TEEs will pave the way for low-cost use-case customization. It will allow the features and the security model to be tuned for each hardware platform and use-case from a set of common software components, drawing on ideas from modular kernel concepts [115, 60, 8]. This motivates the need for *Customizable TEEs*—an abstraction that allows entities that create the hardware, operate it, and develop applications to configure and deploy various TEE designs from the same base. Customizable TEEs promise independent exploration of gaps/trade-offs in existing designs, quick prototyping of new feature requirements, a shorter turn-around time for fixes, adaptation to threat models, and usage-specific deployment.

For realizing this vision, the first observation is the need for a highly programmable trusted layer below the *untrusted* OS. Second, the TEE must decouple the isolation mechanisms from decisions of resource management, virtualization, and trust boundaries. A hypervisor solution results in a trusted layer with a mix of security and virtualization responsibilities, thus complicating the most critical component. Similarly, firmware and micro-code are not programmable to a degree that satisfies the requirements. These two requirements help avoid the mistake of using hardware with a separation mechanism encumbered with a static boundary between what is trusted and untrusted. Lastly, this thesis draw inspiration from proliferation of commercial (c.f. Intel SGX, TrustZone) and non-commercial TEEs (c.f. Sanctum [50], Komodo [65]) which demonstrate the need for a common, portable software base adaptable to ever-changing hardware capabilities and use-case demands.

To this end, this chapter proposes Keystone—the first open-source framework for building customized TEEs. Keystone is based on standard RISC-V specifications [217] for *physical memory protection* (PMP)—a primitive which allows the programmable *machine mode* underneath the OS in RISC-V to specify arbitrary protections on physical memory regions. Keystone uses this machine mode to execute a trusted *security monitor (SM)* to provide security boundaries without needing to perform any resource management. Critically, each enclave operates in its own isolated physical memory region and has its own supervisor-mode *runtime* (RT) component to manage the virtual memory of the enclave and more. With this novel design, any enclave-specific functionality can be implemented cleanly by its RT while the SM manages hardware-enforced guarantees. An enclave's RT implements only the required functionality, communicates with the SM, mediates communication with the host via shared memory, and services the *enclave user-mode application (eapp)*.

The choice of RISC-V and the logical separation between SM and RT allows hardware manufacturers, cloud providers, and application developers to configure various design choices such as TCB, threat models, workloads, and TEE functionality. Specifically, Keystone's SM uses hardware primitives to provide in-built support for TEE guarantees such as measured boot, memory isolation, and attestation. The RT then provides functionality modules for system call interfaces, standard `libc` support, in-enclave virtual memory management, self-paging, and more inside the enclave. For strengthening the security, the SM leverages any available configurable hardware to compose additional security mechanisms. Later sections demonstrate the potential of this with a highly configurable cache controller to, in concert with PMP, transparently defend against physical adversaries and cache side-channels.

The implementation of this chapter includes the SM, two RTs (the native RT—Eyrie—and an off-the-shelf microkernel seL4 [101]), and several modules which together allow enclave-bound user applications to selectively configure and use the above features (Figure 4.1). Section 4.7 extensively benchmarks Keystone on 4 suites with varying workloads: RV8, IOZone, CoreMark, and Beebs. The section showcases use-case studies where Keystone can be used for secure machine learning (Torch and FANN frameworks) and cryptographic tasks (`libsodium`) on embedded devices and cloud servers. Lastly, Keystone is tested on different RISC-V systems: the HiFive Freedom Unleashed, 3 in-order cores and 1 out-of-order core via FPGA, and a QEMU emulation—all without modification. Keystone is fully open-source.

This chapter consists of the following contributions:

- *Customizable TEEs.* Define a new paradigm wherein the hardware manufacturer, hardware operator, and the enclave programmer can tailor the TEE design.
- *Keystone Framework.* Present the first framework to configure, build, and instantiate customized TEEs. The principled way of ensuring modularity in Keystone allows one to customize the design dimensions of TEE instances as per the requirements.
- *Open-source Implementation.* Demonstrate advantages of different Keystone TEE configurations that are tailored for minimizing the TCB, adapting to threat models, using hardware features, handling workloads, or providing rich functionality without any micro-architectural changes. A typical Keystone instantiated TEE design adds a total TCB of 12-15 K lines of code (LoC) to an enclave-bound application, of which the SM consists of only 1.6 KLoC added by Keystone.
- *Benchmarking & Real-world Applications.* Evaluate Keystone on 4 benchmarks: CoreMark, Beebs, and RV8 ($< 1\%$ overhead), and IOZone (40%). The evaluation demonstrates real-world machine learning workloads with Torch in Eyrie (7.35%), FANN (0.36%) with seL4, and a Keystone-native secure remote computation application. Finally, the evaluation demonstrates defenses against physical adversaries with memory encryption and cache side-channels.

4.2 What is a Common Base for Diverse TEEs?

4.2.1 Background: Commercial TEEs

Current widely-used TEE systems cater to specific and valuable use-cases but occupy only a small part of the wide design space (see Table 4.1). Consider the case of a heavy server workload (databases, ML inference, etc.) running in an untrusted cloud environment. One option is an Intel SGX-based solution which has a large software stack [24, 209, 15] to extend the supported features. On the other hand, an AMD SEV-based solution isolates a full VM with a large TCB. If one wants additional defenses against side-channels it adds further user-space software mechanisms for both cases. If one considers edge-sensors or IoT applications, the available solutions are TrustZone based. While more flexible than SGX or SEV, TrustZone supports only a single hardware-enforced isolated domain called the Secure World. Any further isolation needs multiplexing between secure

	System	**C3. Software Adversary**	**C4. Physical Adversary**	**C5. Side-Channel Adversary**	**C6. Ctrl-Channel Adversary**	**C7. Low Software TCB**	**C8. No Hardware Modification**	**C9. Resource Management**	**C10. All Applications**	**C11. High Expresiveness**	**C12. Low Porting Efforts**
Intel	SGX [126]	●	●	○	○	●	○	○	○	○	○
	Haven [24]	●	●	○	○	○	●	◐	◐	●	●
	Graphene-SGX [209]	●	●	○	○	○	●	◐	◐	●	●
	Scone [15]	●	●	○	○	◐	●	○	◐	●	◐
ARM	TrustZone [13]	●	○	○	○	○	○	●	●	●	○
	Komodo [65]	●	●	○	●	◐	●	◐	●	◐	●
	OP-TEE [146]	●	●	○	○	○	●	●	●	●	●
	Sanctuary [31]	●	○	◐	○	○	●	◐	●	●	●
AMD	SEV [6]	◐	◐	○	○	●	○	●	●	●	●
	SEV-ES [96]	●	●	○	○	●	○	●	●	●	●
RISC-V	Sanctum [50]	●	○	●	●	◐	○	◐	◐	◐	●
	TIMBER-V [171]	●	●	○	○	●	○	●	◐	●	◐
	MultiZone [134]	●	●	●	●	◐	●	○	○	○	◐
	Keystone (This thesis)	●	●	●	●	◐	●	●	●	◐	●

Table 4.1: Trade-offs in existing TEEs/extensions. ●, ◐, ○: best to worst respectively. C3-6: resilience to software adversary, hardware adversary, side-channel adversary, controlled-channel adversary respectively. indicates complete protection; confidentiality only; no protection. C7: zero; thousands LoC; millions LoC. C8: zero; non-zero hardware; micro-architectural modifications. C9: enclave self resource management; partial; no flexibility. C10: range of apps supported are maximum; specific class; only written from scratch. C11: expressiveness includes forking, multi-threading, syscalls, shared memory; partial; none of these. C12: dev-effort for porting is unmodified binaries; compiling and/or configuration files; re-writing.

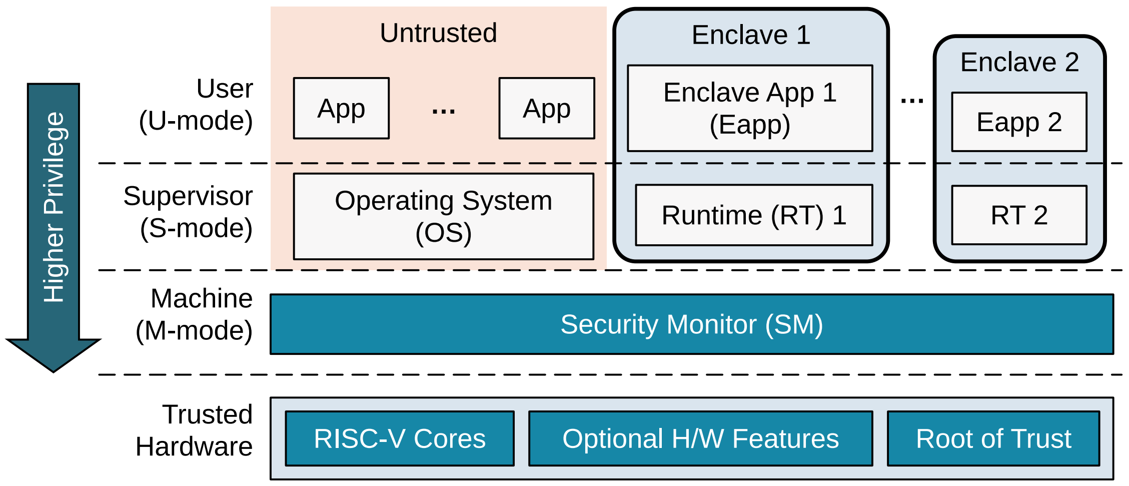

Figure 4.1: Keystone system with host processes, untrusted OS, security monitor, and multiple enclaves (each with runtime and eapp)

applications via software-based Secure World OS solutions [146]. Thus, irrespective of the TEE, developers often compromise their requirements (e.g., resort to a large TCB solution, one isolation domain) or build their custom design. One such emerging direction is to use a thin layer of trusted software, similar to a reference monitor in kernel designs. These designs protect against a strong adversary and provide better compatibility while maintaining a low TCB. Several proposals in this area have demonstrated the feasibility of this approach. Sanctum uses a series of modifications to hardware to construct user-space enclaves for RISC-V. Komodo takes this concept further and provides a verified monitor that executes on top of ARM's TrustZone. While these systems inherit the limitations of their underlying designs (e.g., hardware changes or only two security domains in TrustZone), monitor-based TEEs are a very promising direction.

4.2.2 Customizable TEEs

This model is called *customizable TEEs*. It uses a common software framework to assemble a specialized TEE specific to the use-case with multiple stakeholders' inputs. The hardware manufacturer is only required to provide basic primitives. Realizing a specific TEE instance involves the platform provider's choice of the hardware interface, the trust model, and the enclave programmer's feature requirements. The entities offload their choices to a framework that composes the required modules to instantiate a specialized TEE.

A motivation for customizable TEEs is that the threat model may differ depending on the use case, the application or the hardware platform. Even on the same platform with the same SM, different applications may operate under differing threat models. For this reason, each enclave should be able to specifiy its configuration of security features. Consider a simple IoT sensor platform that signs measurements for authenticity guarantees and an adversary using a cache occupancy side-channel. In this case, the sensor driver must be protected and requires runtime memory integrity, but not memory confidentiality. The signing process requires both memory integrity and confidentiality. Thus, a possible configuration would be to have the cryptographic library operate with a private cache partition enclave while the driver may operate in a basic isolated enclave. An appropriate SM mechanism (e.g. mailboxes) can ensure authenticated communication between these two enclaves. An adversary using a cache occupancy side-channel against the driver learns only the public measurements, and cannot learn anything about the cryptographic library. Allowing each enclave to specify and deploy its own defenses can optimize the use of the available resources (in this case, limited private cache space) and expensive security mechanisms.

The existing commercial TEE systems offer inflexible threat models linked to the respective hardware platform. Notably, Intel's SGX [126] does not support any configuration of its memory protection systems as would be desirable for use cases not requiring expensive memory encryption. On the other hand, while offering some software and hardware customization, ARM's TrustZone provides an inferior substrate to build a modular TEE. Core to TrustZone's design is the concept of only two security domains. A TrustZone TEE implementing multiple enclaves must use the memory management unit (MMU) for further isolation. This fundamentally limits what operations enclaves can be allowed to perform and limits enclaves to user-mode. This limitation naturally extends to all TEE systems built using TrustZone as a base like Komodo. On the hardware side, TrustZone relies on system-wide bus-address filters (e.g., the TZC-400) to separate secure from insecure DRAM partitions, whereas RISC-V provides per-hardware-thread views of physical memory via machine-mode and PMP registers. Using RISC-V thus allows multiple concurrent and potentially multi-threaded enclaves to access disjoint memory partitions while also opening up supervisor-mode and the MMU for enclave use. This allows an enclave to contain either a lightweight or even a full supervisor-mode OS as demonstrated in the later sections.

Keystone requires no changes to CPU cores, memory controllers, etc. A secure hardware platform supporting Keystone requires: a device-specific secret key visible only to the trusted boot process, a hardware source of randomness, and a trusted boot process. Key provisioning [9] is an orthogonal problem. Keystone assumes a simple manufacturer provisioned key.

4.2.3 Entities in TEE Lifecycle

There are five logical entities in customizable TEEs:

- *Hardware manufacturer* designs and fabricates RISC-V hardware including relevant IP for trusted boot.
- *Keystone platform provider* purchases manufactured hardware; operates the hardware; makes it available for use to its customers; configures the SM.
- *Keystone programmer* develops Keystone software components including SM, RT, and eapps. Keystone programmer are the respective programmers who develop these specific components.

- *Keystone user* chooses a Keystone configuration of RT and an eapp. They instantiate an enclave which can execute on hardware provisioned by the Keystone platform provider.
- *Eapp user* interacts with the eapp executing in an enclave on the TEE instantiated using Keystone.

In real-world deployments, a single entity can perform multiple roles. For example, consider Acme Corp. hosts their website on an Apache webserver executing on Bar Corp. manufactured hardware in a Keystone-based enclave hosted on Cloud Corp. cloud service. In this scenario, Bar will be the Hardware manufacturer; Cloud will be a Keystone platform provider and can be an RT programmer and SM programmer; Apache developers will be eapp programmer; Acme Corp. will be Keystone user, and; the person who uses the website will be the eapp user.

4.3 Keystone Design Overview

Keystone is based on RISC-V, which is an open ISA with multiple open-source core implementations [16, 38]. It currently supports up to four privilege modes: U-mode (user) for user-space processes, S-mode (supervisor) for the kernel, H-mode (hypervisor) for the hypervisor, and M-mode (machine) which directly accesses physical resources (e.g., interrupts, memory, devices). At the time of writing, H-mode (hypervisor) is not included in the standard specification. Keystone will also be able to support hypervisor-level isolation when H-mode becomes available.

4.3.1 Design Principles

Customizable TEEs can increase flexibility and reduce effort using the following principles.

1. ***Leverage programmable layer and isolation primitives below the untrusted code.*** A reference monitor style *security monitor* (SM) enforces TEE guarantees on the platform using four properties of M-mode: (a) it is programmable by platform providers, (b) it meets the needs for a minimal highest privilege mode, (c) it controls hardware delegation of the interrupts and exceptions in the system, and (d) M-mode's control of RISC-V's Physical Memory Protection (PMP) standard [217] enables isolation of memory-mapped control features at runtime.

2. ***Decouple the resource management and security checks.*** The SM enforces security policies with minimal code at the highest privilege. It has few non-security responsibilities. This keeps the TCB low and allows it to present clean abstractions. The S-mode runtime (RT) and U-mode enclave application (eapp) both reside in enclave address space and are isolated from the untrusted OS or other user applications. The RT manages the lifecycle of the user code executing in the enclave, manages memory, services syscalls, etc. For communication with the SM, the RT uses a limited set of API functions via the RISC-V supervisor binary interface (SBI) to exit or pause the enclave (Table 4.2) as well as request SM operations on

behalf of the eapp (e.g., attestation). Each enclave instance may choose its own RT which is never shared with any other enclaves.

3. ***Design modular layers.*** Keystone uses modularity (SM, RT, eapp) to support a variety of workloads. It frees Keystone platform providers and Keystone programmers from retrofitting their requirements and legacy applications into an existing TEE design. Each layer is independent, provides a security-aware abstraction to the layers above it, enforces guarantees which can be easily checked by the lower layers, and is compatible with existing notions of privilege.

4. ***Allow fine-grained TCB configuration.*** Keystone can instantiate TEEs with the minimal TCB for given specific use-cases. The enclave programmer can further optimize the TCB via RT choice and eapp libraries using existing user/kernel privilege separation. For example, if the eapp does not need libc support or dynamic memory management, Keystone will not include them in the enclave.

4.3.2 Keystone Enclave Workflow

Figure 4.2 details the steps from Keystone provisioning to eapp deployment. The platform provider instantiates a SM with a proper hardware specification and security extenstions that bring additional isolation guarantees such as cache partitioning. Independently, the enclave developers use Keystone tools and libraries to write eapps and RT with rich features such as virtual memory management and system calls. The RT may use available SM SBI call, but they do not change the isolation guarantees that the SM enforces.

4.3.3 Writing eapps

Keystone supports 3 ways of writing enclave applications as: (a) standalone Keystone-native eapps, (b) un-modified RISC-V binaries with RT support, or (c) partitioned applications running selected parts in the enclave. Future work will allow Keystone to operate as a backend for cross-enclave SDKs (e.g., OpenEnclave [144], Asylo [158]) to allow for a wide variety of programming models. Sections 4.7.4, 4.7.3 demonstrate un-modified RISC-V binaries and a manual partitioning.

4.3.4 Threat Model

The Keystone framework trusts the PMP specification as well as the PMP and RISC-V hardware implementation to be bug-free. The Keystone user trusts the SM only after verifying if the SM measurement is correct, signed by trusted hardware, and has the expected version. The SM only trusts the hardware, the host trusts the SM, the RT trusts the SM, the eapp trusts the SM and the RT.

Keystone can operate under diverse threat models, each requiring different defense mechanisms. For this reason, this section outlines all relevant attackers for Keystone. Keystone allows

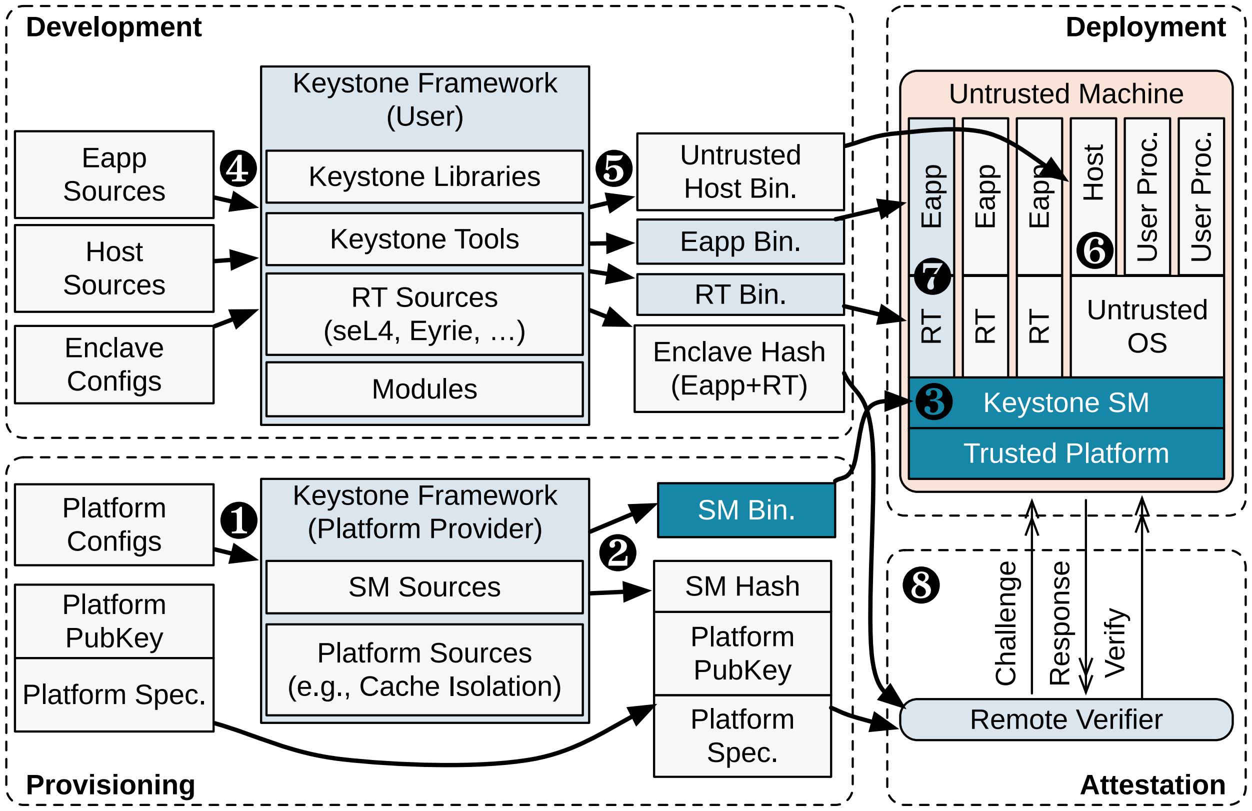

Figure 4.2: Keystone End-to-end Overview. ❶ Platform provider configures the SM. ❷ Keystone compiles and generates the SM boot image. ❸ Platform provider deploys the SM. ❹ Developer writes an eapp, configures the enclave. ❺ Keystone builds the binaries, computes measurements. ❻ Untrusted host binary is deployed to the machine. ❼ Host deploys the RT, the eapp, and initiates the enclave creation. ❽ Remote verifier can attest based on known platform specifications, keys, and SM/enclave measurements.

the selection of a sub-set of these attackers based on the scenario. For example, if the user is deploying TEEs in their private data centers or home appliances, a physical attacker may not be a realistic threat and Keystone can be configured to operate without physical adversary protections.

Attacker Models. Keystone protects the confidentiality and integrity of all enclave code and data at all times after creation. This section defines four classes of attackers who aim to compromise the security guarantees:

- A *physical attacker* A_{Phy} can intercept, modify, or replay signals that leave the chip package. The physical attacker does not affect the components inside the chip package. A_{Phy_C} is for confidentiality, A_{Phy_I} is for integrity.
- A *software attacker* A_{SW} can control host applications, the untrusted OS, network communications, launch adversarial enclaves, arbitrarily modify any memory not protected by the TEE, and add/drop/replay enclave messages.
- A *side-channel attacker* A_{SC} can glean information by observing interactions between the trusted and the untrusted components via the cache side channel (A_{Cache}), the timing side channel (A_{Time}) or the controlled channel (A_{Cntrl}).
- A *denial-of-service attacker* A_{DoS} can take down the enclave or the host OS. Keystone allows the OS to DoS enclaves as the OS can refuse services to user applications at any time.

Scope. Keystone currently has no meaningful mechanisms to protect against speculative execution attacks [102, 35]. Existing and future defenses against this class of attacks can be retrofitted into Keystone [29, 221]. Keystone does not natively protect enclaves or the SM against timing side-channel attacks. Programmers should use existing software solutions to mask timing channels [68] and hardware manufacturers can supply timing side-channel resistant hardware [100]. Side-channel attacks with off-chip components (e.g., memory bus [109]) are also out-of-scope of this thesis and they can be orthogonally mitigated by oblivious memory. The SM exposes a limited API (i.e., SBI) to the host OS and the enclave. Keystone does not provide non-interference guarantees for this API [65]. Similarly, the RT can optionally perform untrusted system calls into the host OS. Keystone assumes that the RT and the eapp have sufficient checks in place to detect Iago attacks via this untrusted interface [187, 149, 209]. Keystone requires that the SM, RT, and eapp are bug-free. This is a strong assumption but can be partially achieved with formal verification [65, 136].

4.4 Security Monitor Design for Multiple Threat Models

The core of a Keystone TEE is the Security Monitor (SM). As the SM uses only standard RISC-V features, it is easily portable to the other RISC-V platforms. In addition, Keystone provides an easy way of configuring and compiling the SM depending on the underlying platform. With this design, Keystone integrates with optional hardware to provide additional security guarantees such as cache side-channel defenses without any application changes. By design, the SM enforces isolation and provides security-critical features without the burden of high-level functionality like virtual memory management. This allows for a simple, and low attack surface, highest-privilege component.

4.4.1 Memory Isolation

Keystone only requires the RISC-V hardware to provide simple security primitives, assigns resource management logic either to the untrusted software or the enclave, and relies on trusted software executing at the highest privilege (e.g., bootloader, SM) to safely validate their decisions.

Keystone uses physical memory protection (PMP), a feature provided by RISC-V. PMP restricts the physical memory access of S-mode and U-mode to certain regions defined via *PMP entries* (See Figure 4.3). Each PMP entry controls the U-mode and S-mode permissions to a customizable region of physical memory.[1] The PMP address registers encode the address of a contiguous physical region, configuration bits specify the r-w-x permissions for U/S-mode, and two addressing mode bits. PMP has three addressing modes to support various sizes of regions (arbitrary regions and power-of-two aligned regions). PMP entries are statically prioritized with the lower-numbered PMP entries taking priority over the higher-numbered entries. If U- or S-mode attempts to access a physical address and it does not match any PMP address range, the hardware does not grant any access permissions.

PMP makes Keystone memory isolation enforcement flexible in three ways: (a) multiple discontiguous enclave memory regions can coexist instead of reserving one large memory region shared by all enclaves, (b) PMP entries can cover regions from 4 bytes to all of DRAM allowing for arbitrarily sized enclaves, (c) PMP entries can be reconfigured during execution to dynamically create new regions or release a region to the OS.

During the SM boot, Keystone configures the first PMP entry (highest priority) to cover its own memory region (code, stack, data such as enclave metadata and keys), disallowing access to it from U-mode and S-mode. It then configures the last PMP entry (lowest priority) to cover all memory and with all permissions enabled to allow the OS default access to memory not otherwise covered by a PMP entry.

When a host application requests the OS to create an enclave, the OS finds an appropriate contiguous physical region[2] and then calls into the SM. After validating the request, the SM protects the enclave memory by adding a PMP entry with all permissions disabled. Since the enclave's PMP entry has a higher priority than the OS PMP entry (the last in Figure 4.3), the OS and other user processes cannot access the enclave region. A valid request requires that enclave regions not overlap with each other or with the SM region.

During control-transfer to an enclave, the SM (for the current core only): (a) enables PMP permission bits of the relevant enclave memory region; and (b) removes all OS PMP entry permissions

[1]Currently processors have up to 64 M-mode configurable PMP entries [217].

[2]The Keystone kernel driver uses both the Buddy Allocator and the Contiguous Memory Allocator (CMA) to dynamically allocate enclave memory with various sizes.

Caller	SM SBI	Description
OS	create	Validate, and measure the enclave
	run	Start enclave and boot RT
	resume	Resume enclave execution
	destroy	Clean & release enclave memory
RT	stop	Pause enclave execution
	exit	Terminate the enclave
	attest	Get a signed attestation report
	random	Get secure random values
OS & RT	extension*	Platform-specific functions

Table 4.2: The SBI functions the SM provides, *SM can provide additional functions (e.g., dynamic resizing) depending on the platform.

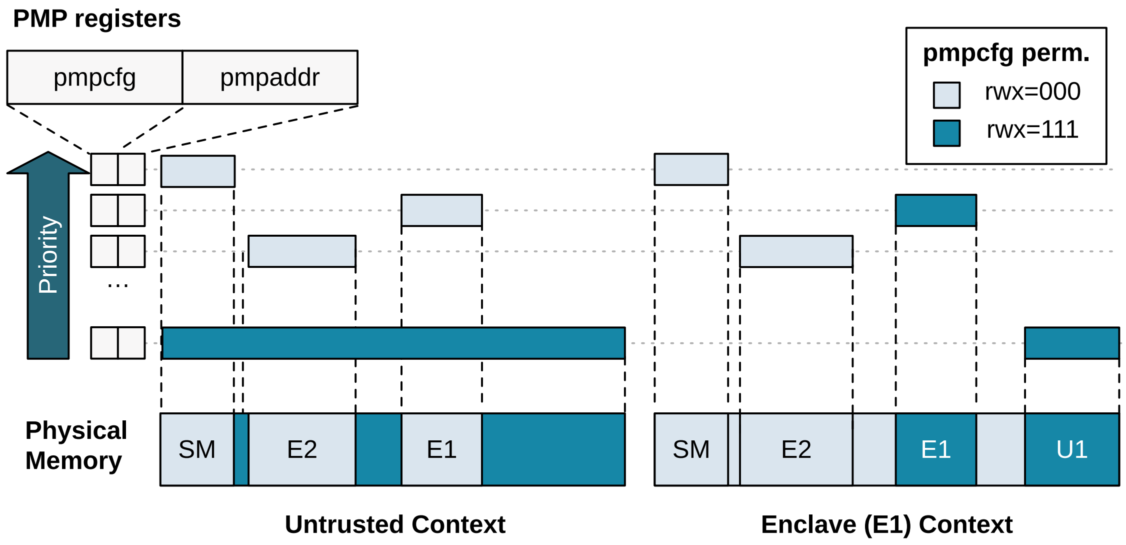

Figure 4.3: How Keystone uses RISC-V PMP for the flexible, dynamic memory isolation. `pmpaddr` and `pmpcfg` control and status registers (CSRs) are used to specify PMP entries. The SM uses a few PMP entries to guard its own memory (`SM`) and enclave memories (`E1`, `E2`). Upon enclave entry, the SM will reconfigure the PMP such that the enclave can only access its own memory (`E1`) and the untrusted buffer (`U1`).

to protect all other memory from the enclave. This allows the enclave to access its own memory and no other regions. At a CPU context-switch to non-enclave, the SM disables all permissions for the enclave region and re-enables the OS PMP entry to allow default access from the OS. Enclave PMP entries are freed on enclave destruction.

Each core has its own complete set of PMP entries. During enclave creation, PMP changes must be propagated to all the cores via inter-processor interrupts (IPIs). The SM executing on each of the cores handles these IPIs by removing the access of other cores to the enclave. During the enclave execution, changes to the PMP entries (e.g., context switches between the enclave and the host) are local to the core executing it and need not be propagated to the other cores. PMP synchronization IPIs are only sent during enclave creation and destruction.

Each allocated enclave (executing or not) requires one PMP entry per isolated memory region it uses. The OS PMP entry is reused during enclave execution for allowing access to shared memory. Additional PMP regions are available to enclaves via SM interfaces and are used for cases like self-paging as described in Section 4.5.1.

Naively, Keystone supports $N - 2$ simultaneously created enclaves, where N is the number of PMP entries available. Alternatively, with adjacent allocations by the OS, Keystone can virtualize the PMP entries at the cost of disallowing memory reclamation until all latter enclaves are destroyed. Future SM and RT features that support relocation may allow for complete virtualization of PMP entries via defragmentation. Similarly, the proposed RISC-V hypervisor mode (H-mode) would allow for an additional layer of address translation to transparently virtualize PMP entries [88].

4.4.2 Post-creation In-enclave Page Management

Keystone has a different memory management design from most TEEs (see Figure 4.4). It uses the OS-generated page tables for initialization and then delegates virtual-to-physical memory mapping entirely to the enclave during execution. Since RISC-V provides per-hardware-thread views of the physical memory via the machine-mode and the PMP registers, it allows Keystone to have multiple concurrent and potentially multi-threaded enclaves to access the disjoint physical memory partitions. With an isolated S-mode inside the enclave, Keystone can execute its own virtual memory management which manipulates the enclave-specific page tables. Page tables are always inside the isolated enclave memory space. By leaving the memory management to the enclave Keystone: (a) allows flexible virtual memory management with several RT modules (see Section 4.5.1); (b) removes controlled side-channel attacks as the host OS cannot modify or observe the enclave virtual-to-physical mapping.

4.4.3 Interrupts and Exceptions

During enclave execution, all machine interrupts trap directly to the SM. Exceptions (e.g. page faults, etc) may be safely delegated to the RT via the RISC-V exception delegation register. The RT then handles exceptions as needed to implement standard kernel abstractions and may forward other traps to the untrusted OS via the SM. To avoid the enclave holding a core to DoS the host the

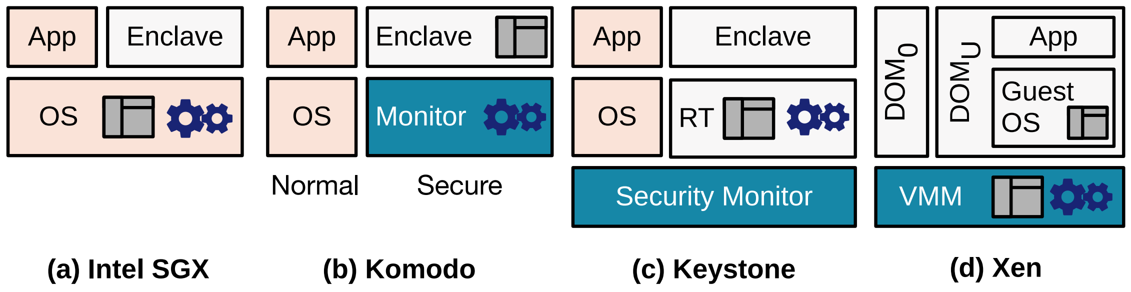

Figure 4.4: Memory Management Designs (red area is untrusted). (a) Untrusted OS manages memory, translates virtual-to-physical address. (b) Page tables inside the enclave but monitor creates mappings. (c) Delegates page management to enclave with its own page table. (d) Hypervisor for page management, 2 page tables.

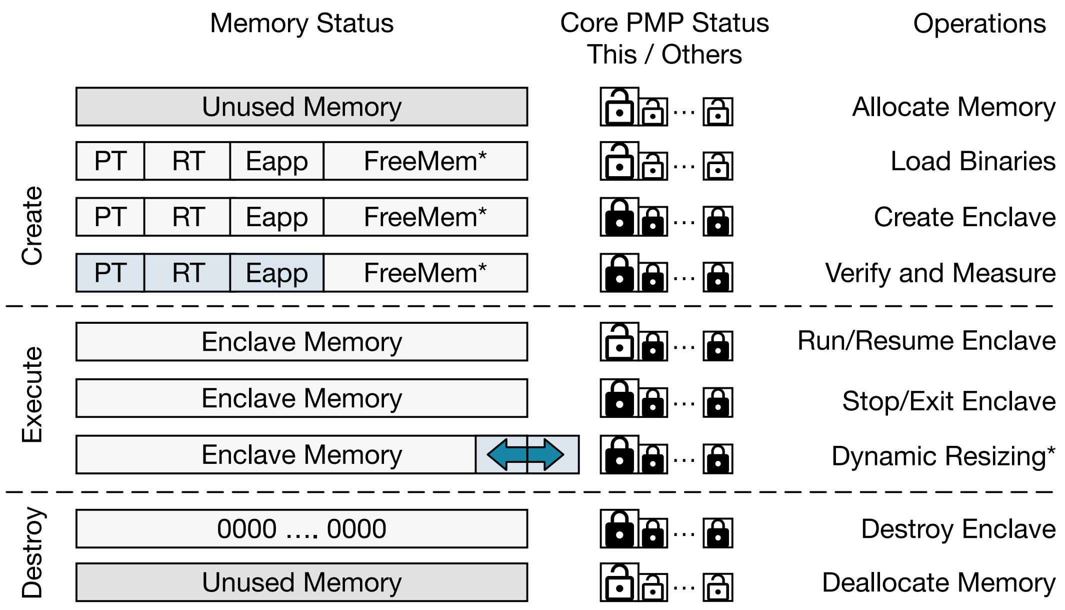

Figure 4.5: Enclave Lifecycle. The enclave memory and the corresponding PMP entry status (accessible or not) are shown per each operation. For PMP status, *This* means the PMP status of the core performing the operation and *Others* is PMP of other cores.

4.4.4 Enclave Lifecycle

SM sets a machine timer before it enters the enclave. When the SM regains control after the timer interrupt triggers, it may return control to the host OS or request that the enclave cleanly exit.

Keystone enclaves go through three distinct changes during their lifecycle as shown in Figure 4.5. At **creation**, Keystone measures the enclave memory to ensure that the OS has loaded the enclave binaries correctly to the physical memory. Keystone uses the initial virtual memory layout for the measurement because the physical layout can legitimately vary (within limits) across different executions. For this, the SM expects the OS to initialize the enclave page tables and allocate physical memory for the enclave. The SM walks the OS-provided page table and checks if there are invalid mappings and ensures a unique virtual-to-physical mapping. The SM then hashes page contents along with the virtual addresses and the configuration data. At **execution**, the SM sets PMP entries and transfers control to the enclave entry point. On an OS initiated **destruction**, the SM clears the enclave memory region before returning the memory to the OS. SM cleans and frees all the enclave resources, PMP entries, and enclave metadata.

4.4.5 TEE Primitives

This section shows some of standard TEE primitives that Keystone supports.

Keystone requires a root-of-trust (RoT), which is a tamper-proof software and hardware component in the platform. A root-of-trust can be implemented in many ways [107, 98] as long as it provides the followings:

- An entropy source that both the RoT and the SM can read
- A platform key store containing a unique device secret provisioned by a trusted hardware vendor
- A tamper-proof measured boot procedure described (Figure 4.6).

Keystone does not rely on a specific implementation. For completeness, currently, Keystone simulates measured boot via a modified first-stage bootloader, with simulated entropy source.

Secure Source of Randomness Keystone provides a secure SM SBI call, `random`, which returns a 64-bit random value. Keystone uses a hardware source of randomness if available or can use other well-known options [133] if applicable.

Measured Boot At each CPU reset, the root-of-trust (a) measures the SM image, (b) generates a fresh attestation key from a secure source of randomness, (c) stores it to the SM private memory, and (d) signs the measurement and the public key with a hardware-visible secret (Figure 4.6). The root of trust must securely pass the generated report to the SM. Finally, the measured boot procedure must have only one exit point, which jumps to the lowest address of the SM. This ensures that the platform always runs the SM that is measured by the root of trust.

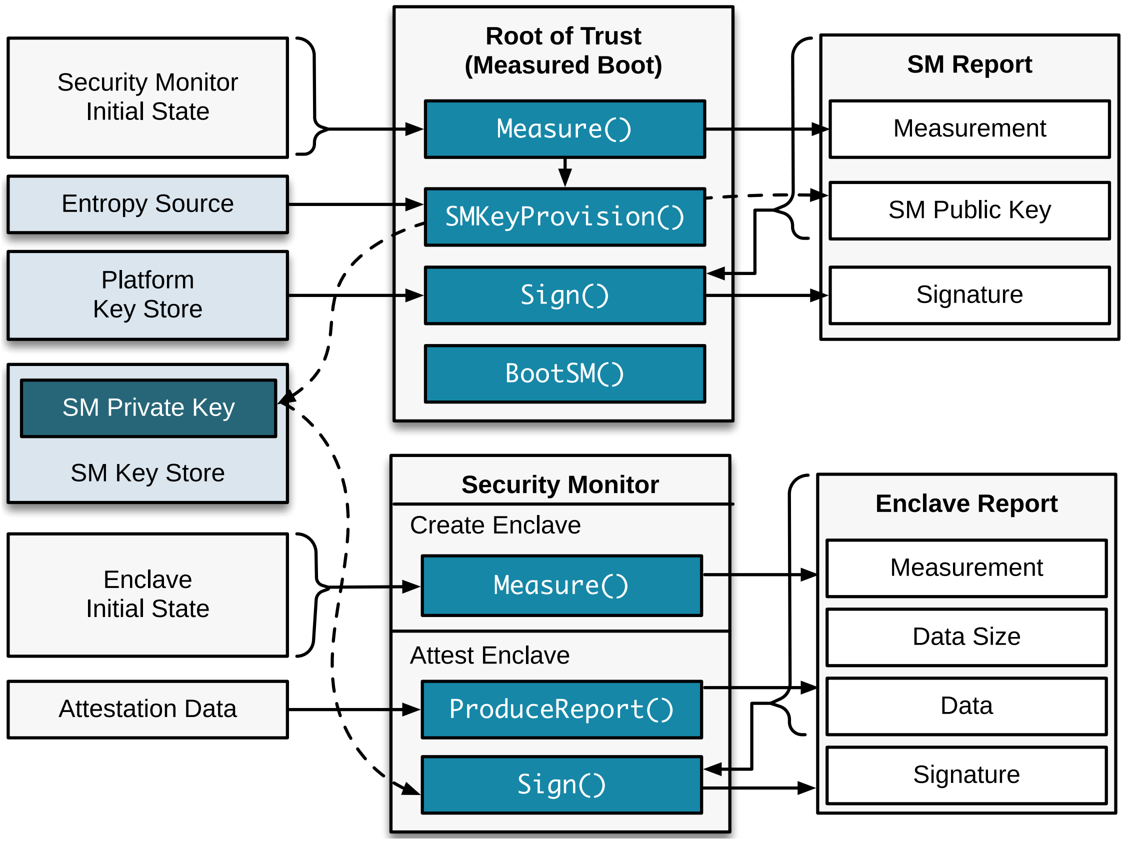

Figure 4.6: Measured boot and attestation in Keystone. All of the root-of-trust functions are trusted, whereas the security monitor functions are trusted given the SM report verification succeeds.

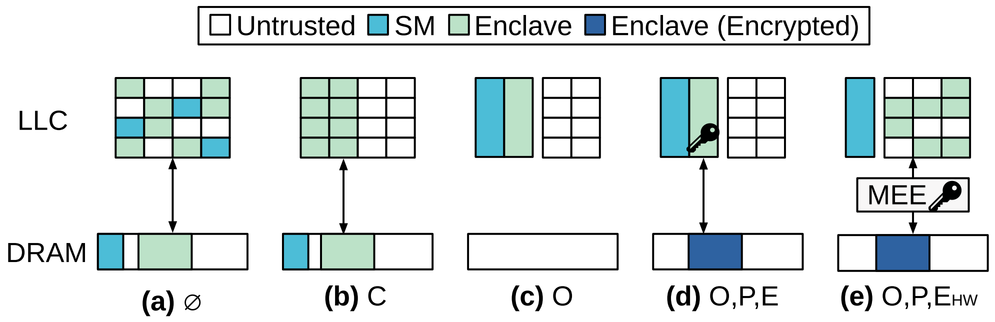

Figure 4.7: Memory Model for Various TEE Scenarios. ∅: baseline, C: cache partitioning, O: on-chip scratchpad, P: enclave self-paging, E: software memory encryption E_{HW}: hardware memory encryption.

Remote Attestation As shown in Figure 4.6, the Keystone SM performs the measurement and the attestation based on the provisioned private key. Enclaves may request a signed attestation from the SM during runtime. The SM produces a report containing the measurement of the enclave, and user-provided arbitrary *attestation data*. The attestation data can be used towards a standard scheme to bind the attestation with a secure channel construction [65, 107]. For example, a program can include freshly-generated Diffie-Hellman (DH) key parameters in the attestation data. Once the enclave report is signed, the report can be passed to a remote verifier such that it can verify the report and complete the key exchange at the same time. Key distribution [9], revocation [83], attestation services [95], and anonymous attestation [34] are orthogonal challenges that this thesis does not cover.

Other Primitives Keystone can support other primitives, if required by the TEE: (a) it allows enclaves to access the read-only hardware-maintained timer registers via the standard `rdcycle` instruction; (b) it can provide monotonic counters by keeping a limited counter state in the SM memory. The SM can implement *trusted timers*, *rollback defense* [49], and *sealed storage* [9] with these features.

4.4.6 Platform-Specific Extensions

Keystone can leverage additional security and functionality features exposed by the hardware to provide stronger security guarantees and/or additional features to the enclave at the cost of various trade-offs. This section demonstrates several examples of customizing the SM for a specific platform so that it can defend the enclave against a physical attacker or cache side-channel attacks. The example in this section uses the HiFive Freedom Unleashed [81] RISC-V dev board containing a Rocket-based quad-core SoC chip (FU540) with a proprietary L2 controller.

Secure On-chip Memory

To protect the enclaves against a physical attacker who has access to the DRAM, Keystone implements an *on-chip memory* extension (Figure 4.7(c)). It allows the enclave to execute without the code or the data leaving the chip package. On the FU540, the SM dynamically instantiate a scratchpad memory of up to 2MB via the L2 memory controller to generate a usable on-chip memory region. The scratchpad is then allocated exclusively to the requesting enclave for it's entire lifetime. An enclave requesting to run in the on-chip memory loads nearly identically to the standard procedure with the following changes: (a) the host loads the enclave to the OS allocated memory region with modified initial page tables referencing the final scratchpad address; and (b) the SM copies the standard enclave memory region into the new scratchpad region before the measurement. Any context switch to the enclave now results in an execution in the scratchpad memory. This uses only the basic enclave life-cycle hooks for the platform-specific features and does not require further modification of the SM. The only other change required was a modification of the untrusted enclave loading process to make it aware of the physical address region that the scratchpad occupies. No modifications to the Eyrie RT or the eapps are required.

Cache Partitioning

Enclaves are vulnerable to cache side-channel attacks from the untrusted OS and other applications via a shared cache. To this end, Keystone implements a cache partitioning scheme using two hardware capabilities: (a) the L2 cache controller's *waymasking* primitive similar to Intel's CAT [138]; (b) PMP to way-partition the L2 cache transparently to the OS and the enclaves. The resulting SM enforces effective *non-interference* between the partitioned domains (Figure 4.7(b)). Upon a context switch to the enclave, the cache lines in the partition are flushed. During the enclave execution, only the cache lines from the enclave physical memory are in the partition and are thus protected by PMP. The adversary cannot insert cache lines in this partition during the enclave execution due to the line replacement way-masking mechanism. As a net effect, adversary (A_{Cache}) gains no information about the evictions, the resident lines, or the residency size of the enclave's cache. Ways are partitioned at runtime and are available to the host whenever the enclave is not executing even if paused.

Dynamic Resizing

Statically pre-defined maximum enclave size and subsequent static physical or virtual memory pre-allocations: (a) prevent the enclave from scaling dynamically based on workload, (b) complicates porting applications to eapps. To this end, Keystone allows the SM to dynamically change the physical memory boundaries of the enclave. The Eyrie RT may request that the OS make an `extend` SBI call to add contiguous physical pages to the enclave memory region. If the OS succeeds in allocating, the SM increases the enclave's size by extending the relevant PMP entries and notifies the RT, which then uses the free memory module to manage the new physical pages (see Section 4.5.1).

4.5 Modular Runtime Design for Extensive Functionality

As the SM physically isolates each of the enclaves, the enclave can safely run private S-mode code (i.e., the RT). This enables modular system-level abstraction for eapps (e.g., virtual memory management). Although the RT is similar in functionality to a kernel inside an enclave, it does not require most kernel functionality. This section presents a modular exemplar RT—**Eyrie**—to allow enclave developers the ability to include only necessary functionality and reduce the TCB.

Given the supervisor capability, implementing selected kernel functionality does not require modifying user applications. The additional privilege layer allows for further defensive design, such as only allowing the RT access to the shared memory buffer. Moreover, it enables easy porting of a full-fledged off-the-shelf microkernel such as seL4 in an enclave. This section introduces key Keystone RT modules and show how they support various workloads with small TCB.

4.5.1 Enclave Memory Management Modules

Each enclave can run both S-mode and U-mode code. Since they have the privileges to manage their own memory, they need not cross the host-enclave isolation boundary. By default, Keystone enclaves occupy a fixed contiguous physical memory allocated by the OS with a statically-mapped virtual address space at load time. While suitable for some embedded applications, it limits the memory usage of most legacy applications. To this end, Keystone implements several optional modules to enable flexible enclave memory management.

Free memory

The *free memory module* allows the Eyrie RT to perform page table management, after the enclave reserves unmapped physical memory. Thus, the page mappings need not be pre-defined at creation time. The unmapped (hence, free) memory region is not included in the enclave measurement and is zeroed before beginning the eapp execution. The free memory module is required for other more complex memory modules.

In-Enclave Self Paging

A module implements a generic in-enclave page swapping for the Eyrie RT. It handles the enclave page-faults and uses a generic page backing-store that manages the evicted page storage and retrieval. The module uses a simple random eapp-only page eviction policy. It works in conjunction with the free memory module for virtual memory management in the Eyrie RT. Put together, they help to alleviate the tight memory restrictions an enclave may have due to the limited DRAM or the on-chip memory size [148, 143, 149].

Protecting the Page Content Leaving the Enclave

When the enclave handles its own page fault, it may attempt to evict the pages out of the secure physical memory (either an on-chip memory or the protected portion of the DRAM). When these pages have to be copied out, their content needs to be protected. Thus, as part of the in-enclave page management, Keystone implements a backing-store layer that can include page encryption and integrity protection to allow for the secure content to be paged out to the insecure storage (DRAM regions or disk). The protection can be done either in the software as a part of the Keystone RT (Figure 4.7(d)) or by a dedicated trusted hardware unit—a memory encryption engine (MEE) [79]—with the SM's on-chip memory capability (Figure 4.7). Admittedly, this incurs significant design challenges in efficiently storing the metadata and performance optimizations. The amount of available on-chip memory for integrity data storage will cap the total possible size of the enclave. Keystone design is agnostic to the specific integrity schemes and can reuse the existing mechanisms [129, 168].

4.5.2 Functionality Modules

This section demonstrates various functionality modules in Eyrie.

Edge Call Interface

The eapp cannot access the non-enclave memory in Keystone. If it needs to read/write the data outside the enclave, the Eyrie RT performs *edge calls* on its behalf. The edge call, which is functionally similar to RPC, consists of an index to a function implemented in the untrusted host application and the parameters to be passed to the function. Eyrie tunnels such a call safely to the untrusted host, copies the return values of the function back to the enclave, and sends them to the eapp. The copying mechanism requires Eyrie to have access to a buffer shared with the host. To enable this: (a) the OS allocates a shared buffer in the host memory space and passes the address to the SM at enclave creation; (b) the SM passes the address to the enclave so the RT may access this memory; (c) the SM uses a separate PMP entry to enable OS access to this shared buffer. All the edge calls have to pass through the Eyrie RT as the eapp does not have access to the shared memory virtual mappings. This module can be used to add support for syscalls, IPC, enclave-enclave communication, and so on. As the current edge interface is a straight-forward shared memory region, it can easily to use alternative methods for dispatching calls such as mailboxes or HotCalls [218].

Keystone allows the *proxying of syscalls* from the eapp to the host application by re-using the edge call interface. The user host application then invokes the syscall on an untrusted OS on behalf of the eapp, collects the return values, and forwards them to the eapp. Keystone can utilize existing defenses to prevent Iago attacks [41] via this interface [187, 149, 209]. Keystone resolves appropriate calls as *in-enclave syscalls* (e.g., mmap, brk, getrandom). Such calls are handled in Eyrie and invoke SM interfaces as needed (e.g. getrandom) before returning to the eapp.

Multi-threading

Keystone runs multi-threaded eapps by delegating the thread management to the runtime. Eyrie RT does not support parallel multi-core enclave execution yet, but this can be implemented by allowing the SM to invoke enclave execution multiple times in different cores.

4.6 Security Analysis

This section argues the security of the enclave, the OS, and the SM based on the threat model outlined in Section 4.3.4.

4.6.1 Protection of the Enclave

Keystone attestation ensures that any modification of the SM, RT, and the eapp is visible while creating the enclave. During the enclave execution, any direct attempt by an A_{SW} to access the enclave memory (cached or uncached) is defeated by PMP. All enclave data structures can only be modified by the enclave or the SM, both are isolated from direct access. Subtle attacks such as controlled side channels (A_{Cntrl}) are not possible in Keystone as enclaves have dedicated page management and in-enclave page tables. This ensures that any enclave executing with any Keystone instantiated TEE is always protected against the above attacks.

Mapping Attacks

The RT is trusted by the eapp, does not intentionally create malicious virtual to physical address mappings [82] and ensures that the mappings are valid. The RT initializes the page tables either during the enclave creation or loads the pre-allocated (and SM validated) static mappings. During the enclave execution, the RT ensures that the layout is not corrupted while updating the mappings (e.g., via mmap). When the enclave gets new empty pages, say via the dynamic memory resizing, the RT checks if they are safe to use before mapping them to the enclave. Similarly, if the enclave is removing any pages, the RT scrubs their content before returning them to the OS.

Syscall Tampering Attacks

If the eapp and the RT invoke untrusted functions implemented in the host process and/or execute the OS syscalls, they are susceptible to Iago attacks and system call tampering attacks [41, 159]. Keystone can re-use the existing shielding systems [15, 187, 209] as RT modules to defend the enclave against these attacks.

Side-channel Attacks

Keystone thwarts cache side-channel attacks (Section 4.4.6). Enclaves do not share any state with the host OS or the user application and hence are not exposed to controlled channel attacks. The SM performs a clean context switch and flushes the enclave state (e.g., TLB). The enclave can defend itself against explicit or implicit information leakage via the SM or the edge call API with known defenses [188, 189]. Only the SM can see other enclave events (e.g., interrupts, faults), these are not visible to the host OS. Timing attacks against the eapp are out of scope.

4.6.2 Protecting the Host OS

Keystone RTs execute at the same privilege level as the host OS, so an A_{SW} in this case is stronger than in SGX. The host OS is not susceptible to new attacks from the enclave because the enclave cannot: (a) reference any memory outside its allocated region due to the SM PMP-enforced isolation; (b) modify page tables belonging to the host user-level application or the host OS; (c) pollute the host state because the SM performs a complete context switch (TLB, registers, L1-cache, etc.) when switching between an enclave and the OS; (d) DoS a core as the enclave will be interrupted by the machine timer set by the SM such that the SM can return the control to the OS.

4.6.3 Protection of the SM

The SM naturally distrusts all the lower-privilege software components (eapps, RTs, host OS, etc.). It is protected from an A_{SW} because all the SM memory is isolated using PMP and is inaccessible to any enclave or the host OS. The SM SBI is another potential avenue of attack. Keystone's SM presents a narrow, well-defined SBI to the S-mode code. It does not do complex resource management and is small enough to be formally verified [65, 136]. The SM is only a reference monitor, it does not require scheduled execution time, so an A_{DoS} is not a concern. The SM can defend against an A_{Cache} and an A_{Time} with known techniques [68, 100].

4.6.4 Protection Against Physical Attackers

Keystone can protect against a physical adversary via platform features and a proposed modification to the bootloader. Similar to Chen et al. [44], the SM uses a scratchpad to store the decrypted code and data, while the supervisor mode component (Eyrie modules) handles paging enclave content to DRAM when the scratchpad becomes full.

The enclave itself is protected by the combination of the on-chip memory and the RT's paging module, with encryption and integrity protection on the pages leaving the on-chip memory. The page backing-store is a standard PMP protected physical memory region now containing only the encrypted pages, similar in concept to the SGX EPC. This fully guarantees the confidentiality and integrity of the enclave code and data from an attacker with control of DRAM.

The SM should be executed entirely from the on-chip memory. The SM is statically sized and has a relatively small in-memory footprint (< 150Kb). On the FU540, this would involve repurposing a portion of the L2 loosely-integrated memory (LIM) via a modified trusted bootloader.

With these techniques in place, content outside of the chip package is either untrusted (host, OS, etc.) or is encrypted and integrity protected (e.g., swapped enclave pages). Keystone accomplishes this with no application modifications.

4.7 Evaluation

This section aims to answer the following questions in the evaluation:

1. **Modularity:** Is the Keystone framework viable in different configurations for real applications?

2. **TCB:** What is the TCB of a Keystone-instantiated TEE in various deployment modes?

3. **Performance:** How much overhead do simple Keystone TEEs add to eapp execution time?

4. **Real-world Applications:** Does Keystone provide expressiveness with minimal developer efforts for eapps?

4.7.1 Implementation & Experimental Setup

SM was implemented on top of the Berkeley Boot Loader (bbl) [167]. It supports M-mode trap handling, boot, and other features. The SM implements the initialization at boot as well as the SBI specified in Table 4.2. Platform-specific extensions have been implemented with hooks in SBI functions. The SM simulates unavailable hardware primitives such as the random number generator and the root of trust. All modules in Sections 4.4 and 4.5 are available as compile-time options.

For the runtimes, Eyrie RT was written from scratch in C. Memory encryption is done via software AES-128 [2] and integrity protection is partially implemented. seL4 microkernel [101] was ported to Keystone by modifying 290 LoC for boot, memory initialization, and interrupt handling. There is no inherent restriction to these two RTs, thus additional options are expected in the future.

The host user-land interface for interactions with the enclaves is provided via a Linux kernel driver that creates a device endpoint (`/dev/Keystone`). The untrusted host OS (i.e., Linux) launches and manages the enclaves via SBI on behalf of the user, and also manages the enclave ownership and enclave-related OS resources.

Keystone provides several libraries (edge-calls, host-side syscall endpoints, attestation, etc.) in C and C++ for the host, the eapp, and interaction with the driver-provided Linux device. The provided tools generate the enclave measurements (hashes) without requiring RISC-V hardware, customize the Eyrie RT, and package the host application, eapps, and RT into a single binary. A complete top-level build solution generates a bootable Linux image (based on the tooling for the HiFive Freedom Unleashed) for QEMU, FPGA softcores, and the HiFive containing the SM, the driver, and the enclave binaries.

The experiments used four different platforms; the HiFive Freedom Unleashed [81] with a closed-source FU540 (at 1GHz), and three open-source RISC-V processors: small Rocket (Rocket-S), default Rocket [16], and Berkeley Out-of-order Machine (BOOM) [38] (See Table 4.4). The experiment instantiated the open-source processors on cloud FPGAs using FireSim [97] which simulates the cores at 1GHz. The host OS is buildroot Linux (kernel 4.15). All performance evaluation was performed on the HiFive and the data is averaged over 10 runs unless otherwise specified.

4.7.2 Modularity & Support

This section outlines the qualitative measurement of Keystone flexibility in extending features, reducing TCB, and using the platform features. Table 4.3 shows the TCB breakdown of various components (required and optional) for the SM and Eyrie RT. Most of the modifications (e.g., additional edge-call features) require no changes to the SM, and the eapp programmer may enable them as needed. Future additions (e.g., ports of interface shields) may be implemented exclusively in the RT. The implementation also adds support for a new RT by porting seL4 to Keystone and use it to execute various eapps (See Section 4.7.4). Keystone passes all the tests in seL4 suite and incurs less than 1% overhead on average over all test cases. The advantage of an easily modifiable SM layer is noticeable when features require interaction with the core TEE primitives like memory

SM Component	LoC	Runtime Component	LoC
Base	1100	—	1800
Edge-call Handling	30	—	300
Dynamic Memory	70	—	100
Memory Isolation	500	`libc` Environment	50
Cache Partitioning	300	In-enclave Paging	300
Measured Boot	170	Syscalls	450
On-chip Memory	50	Free Memory	300
		IO Syscall Proxying	300

Table 4.3: TCB Breakdowns for the Eyrie RT and SM features in LoC.

isolation. The SM features were able to take advantage of the L2 cache controller on the FU540 to offer additional security protections (cache-partitioning and on-chip isolation) without changes to the RT or eapp.

TCB Breakdown. Keystone comprises of the M-mode components (`bbl` and SM), the RT, the untrusted host application, the eapp, and the helper libraries, of which only a fraction is in the TCB. The M-mode component is 10.7 KLoC: a cryptographic library (4 KLoC), pre-existing trap handling, boot, and utilities (4.7 KLoC), the baseline SM (1.6 KLoC), and platform-specific code for FU540 (400 LoC). A minimum Eyrie RT is 1.8 KLoC, with modules adding further code as shown in Table 4.3 up to a maximum Eyrie RT TCB of 3.6 KLoC. The current maximum TCB for an eapp running on the SM and Eyrie RT is thus a total of 15 KLoC. TCB calculations were made using `cloc` [48] and `unifdef` [211].

4.7.3 Benchmarks

This section uses 4 standard benchmark suites with a mix of CPU, memory, and file I/O for system-wide analysis: Beebs, CoreMark, RV8, and IOZone. This section reports the overheads of the cache partitioning and physical attacker protection with RV8 as an example of Keystone trade-offs. In all the graphs, 'other' refers to the lifecycle costs for enclave creation, destruction, etc. All benchmarks are run as unmodified RISC-V binaries using an Eyrie runtime with relevant modules as needed.

Common Operations

Figure 4.8 shows the breakdown of various enclave operations. Initial validation and measurement dominate the startup with 2M and 7M cycles/page for FU540 and Rocket-S due to an unoptimized software implementation of SHA-3 [201]. The remaining enclave creation time totals 20k-30k cycles. Similarly, the attestation is dominated by the ed25519 [59] signing software implemen-

Platform	**Core**		**Cache Size (KB)**		**Latency (cycles)**		**# of TLB Entries**	
	#	**Type**	**L1-I/D**	**L2**	**L1**	**L2**	**L1**	**L2**
Rocket-S	1	in-order	8/8	512	2	24	8	128
Rocket	1	in-order	16/16	512	2	24	32	1024
BOOM	1	OoO	32/32	2048	4	24	32	1024
FU540	4	in-order	32/32	2048	2	12-15*	32	128

Table 4.4: Hardware specification for each platform. L2 cache latency in FU540 (*) is based on estimation.

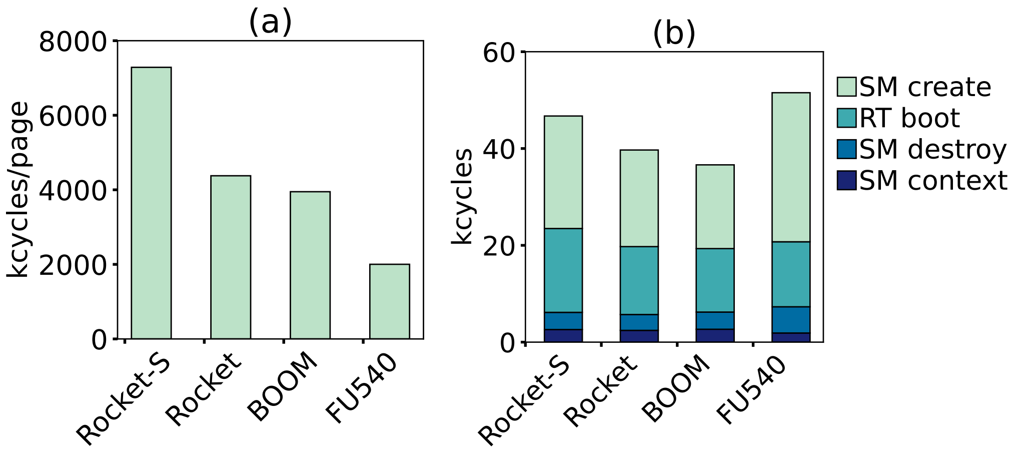

Figure 4.8: Breakdown of operations during the enclave life-cycle. (a) shows enclave validation and hashing duration, and (b) shows the breakdown of other operations. (b) does not include duration of size-dependent operations such as measurement in `create` (Shown in (a)) and memory cleaning in `destroy` (4K-11K cycles/page).

tation (not shown in the graph, 0.7M-1.6M cycles). These are both one-time costs per-enclave and can be substantially optimized in software or hardware. The most common SM operation, context switches, currently take between 1.8K(FU540)-2.6K(Rocket-S) cycles depending on the platform. Notably, creation and destruction of enclaves takes long on the FU540 (4-core) due to the multi-core PMP synchronization.

Standard Benchmarks as Unmodified eapp Binaries

Beebs, CoreMark, and RV8. As expected, Keystone incurs no meaningful overheads ($\pm 0.7\%$, excluding enclave creation and destruction) for pure CPU and memory benchmarks.

IOZone. All the target files are located on the untrusted host and Keystone tunnels the I/O syscalls to the host application. Figure 4.9 shows the throughput plots of common file-content access patterns. Keystone experiences expected high throughput loss for both write (avg. 36.2%) and read (avg. $40.9.\%$). Three factors contribute to the overhead: (a) all the data crossing the privilege boundary is copied an additional time via the untrusted buffer, (b) each call requires the RT to go through the edge call interface, incurring a constant overhead, and (c) the untrusted buffer contends in the cache with the file buffers, incurring an additional throughput loss on re-write (avg. 38.0%), re-read (avg. 41.3%), and record re-write (avg. 55.1%) operations. Since (b) is a fixed cost per system call, it increases the overhead for the smaller record sizes.

Cache Partitioning

The mix of pure-CPU and large working-set benchmarks in **RV8** are ideal to evaluate the impact of caceh partitioning. The experiment granted 8 of the 16 ways in the L2 cache to the enclave during execution (see Figure 4.10). Small working-set tests show low overheads from cache flush on context switches whereas large working-set tests (primes, miniz, aes) show up to 128% overhead due to a smaller effective cache. Enclave initialization latency is unaffected.

Physical Attacker Protections.

The experiment ran the **RV8** suite with on-chip execution, enclave self-paging, page encryption, and a DRAM backing page store (Table 4.5). A few eapps (`sha512`, `dhrystone`), which fit in the 1MB on-chip memory, incur no overhead and are protected even from A_{Phy}. For the larger working-set-size eapps, the paging overhead increases depending on the memory access pattern. For example, `primes` incurs the largest amount of page faults because it allocates and randomly accesses a 4MB buffer causing a page fault for almost every memory access. Software-based memory encryption adds $2-4\times$ more overhead to page faults. These overheads can be alleviated by the Keystone framework if a larger on-chip memory or dedicated hardware memory encryption engine is available as Section 4.5 discussed.

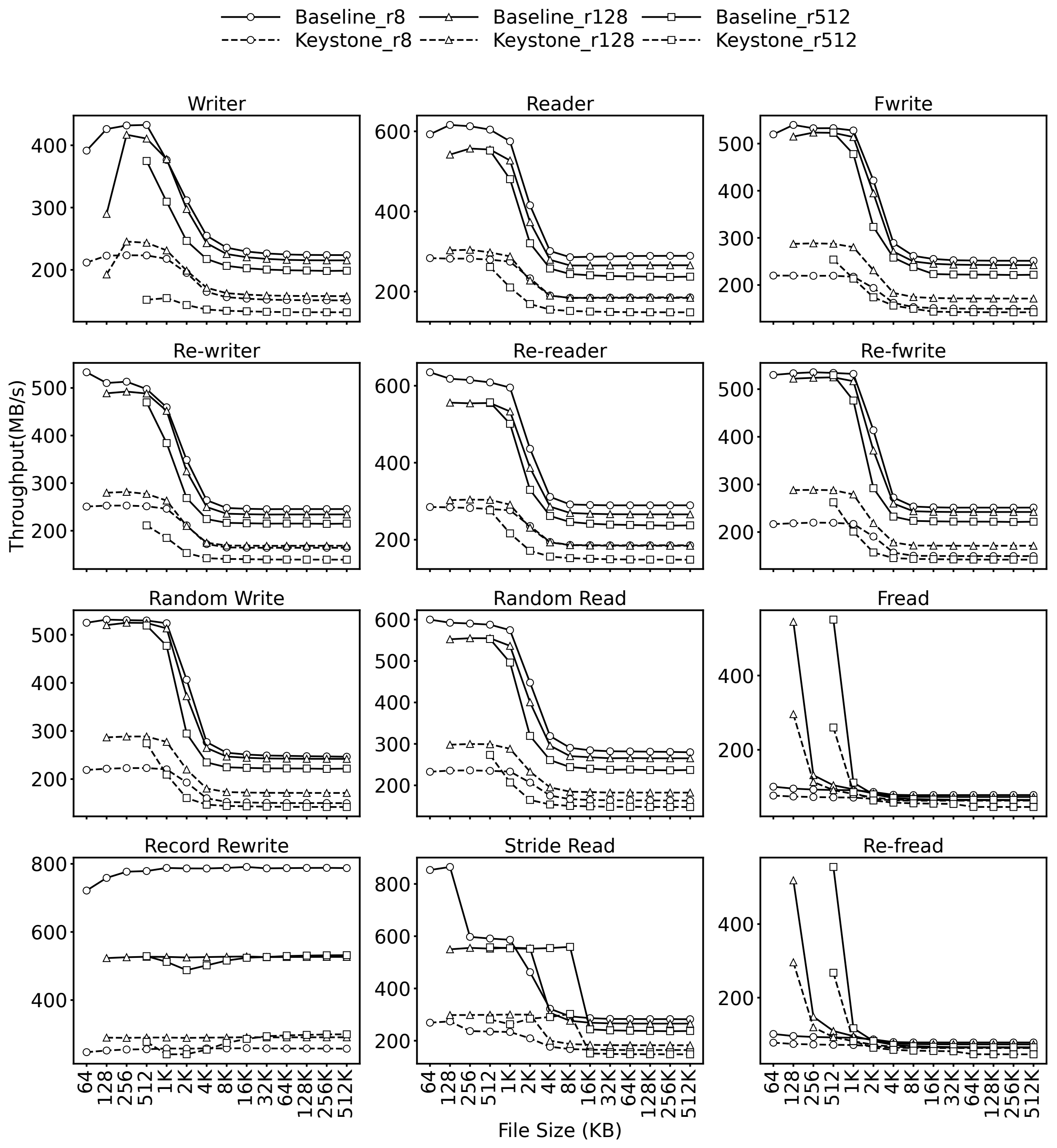

Figure 4.9: IOZone throughput in Keystone for various file and record sizes (e.g., r8 represents 8KB record).

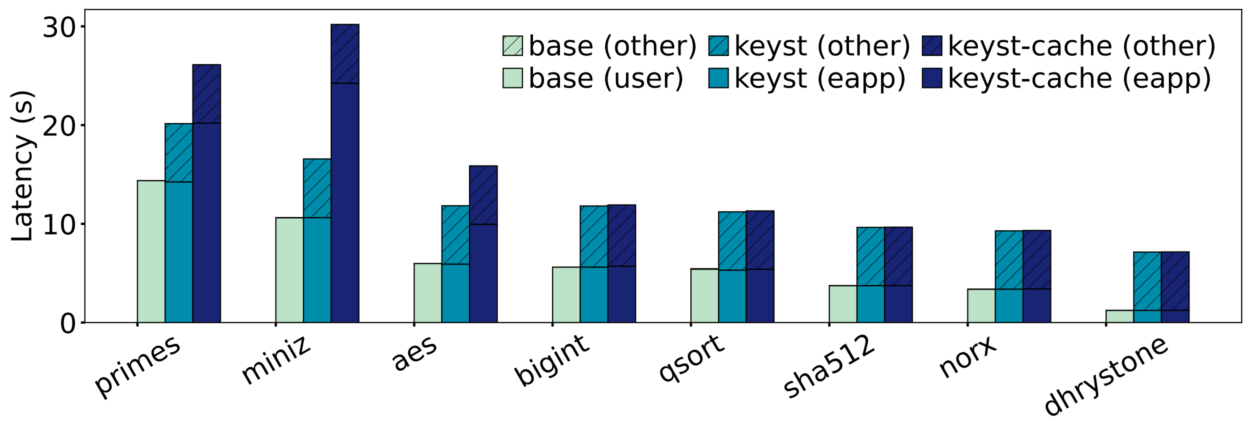

Figure 4.10: Full-execution time comparison for RV8. Each bar shows the duration of the application (`user` or `eapp`), and the other overheads (`other`). Keystone (`keyst`) and Keystone with cache partitioning (`keyst-cache`) compared to native execution (`base`).

	Overhead (%)				**# of Page**
Benchmark	**∅**	**C**	**O, P**	**O, P, E**	**Faults**
primes	-0.9	40.5	65475.5	*	66×10^6
miniz	0.1	128.5	80.2	615.5	18341
aes	-1.1	66.3	1471.0	4552.7	59716
bigint	-0.1	1.6	0.4	12.0	168
qsort	-2.8	-1.3	12446.3	26832.3	285147
sha512	-0.1	0.3	-0.1	-0.2	0
norx	0.1	0.9	2590.1	7966.4	58834
dhrystone	-0.2	0.3	-0.2	0.2	0

Table 4.5: RV8 Overhead for different TEE design instances. ∅: baseline, C: cache partitioning, O: on-chip scratch pad execution (1MB), P: enclave self-paging, E: software-based memory encryption. *: does not complete in ~10 hrs.

4.7.4 Case Studies

This section demonstrates how Keystone can be adapted for a varied set of devices, workloads, and application complexities with three case-studies: (a) machine learning workloads for the client and server-side usage, (b) machine learning for varied RTs, (c) a small secure computation application written natively for Keystone. The evaluation for these case-studies was performed on the HiFive board. The case study used the unmodified application code logic, hard-coded all the configurations and arguments for simplicity, and statically linked the binaries against `glibc` or `musl libc` supported by the Eyrie RT. The widely used cryptographic library `libsodium` can be ported to both Eyrie and seL4 RT trivially.

Case Study 1: Secure ML Inference with Torch and Eyrie

The case study ran nine Torch-based models of increasing sizes with Eyrie on the Imagenet dataset [55] (see Table 4.6). They comprise 15.7 and 15.4KLoC of TH [204] and THNN [203] libraries from Torch compiled with `musl libc`. Each model has an additional 230 to 13.4 KLoC of model-specific inference code [202]. The case study performed two sets of experiments: (a) execute the model inference code with static maximum enclave size; (b) with dynamic resizing support to allow the enclave size to increase on-demand. Figure 4.11 shows the performance overheads for both configurations and non-enclaved execution baseline.
Initialization Overhead is noticeably high for both static size and dynamic resizing. It is proportional to the eapp binary size due to enclave page hashing. Dynamic resizing reduces the initialization latency by 2.9% on average as the RT does not map free memory during enclave creation.
Eapp Execution Overhead was between -3.12%(LeNet) and 7.35%(Densenet) for all the models with both static size and dynamic resizing. The causes of this are: (a) Keystone loads the entire binary in physical memory before it beings eapp execution, precluding any page faults for zero-fill-on-demand or similar behavior, so smaller sized networks like LeNet execute faster in Keystone and (b) the overhead is primarily proportional to the number of layers in the network, as more layers results in more memory allocations and increase the number of `mmap` and `brk` syscalls. A small hand-coded test verified that Eyrie RT's custom `mmap` is slower than the baseline kernel and incurs overheads. Densenet, which has the maximum number of layers (910), thus suffers from larger performance degradation. In summary, for long-running eapps, Keystone incurs a fixed one-time startup cost and the dynamic resizing is indeed useful for larger eapps.

Case Study 2: Secure ML with FANN and seL4

Keystone can be used for small devices such as IoT sensors and cameras to train models locally as well as flag events with model inference. The case study ran FANN, a minimal (8KLoC C/C++) eapp for embedded devices with the seL4 RT to train and test a simple XOR network. The end-to-end execution overhead is 0.36% over running in seL4 without Keystone.

Model	**# of Layers**	**# of Param**	**App LOC**	**Binary Size**	**Memory Usage**
Wideresnet	93	36.5M	1625	140MB	384MB
Resnext29	102	34.5M	1910	123MB	394MB
Inceptionv3	313	27.2M	5359	92MB	475MB
Resnet50	176	25.6M	3094	98MB	424MB
Densenet	910	8.1M	13399	32MB	570MB
VGG19	55	20.0M	1088	77MB	165MB
Resnet110	552	1.7M	9528	7MB	87MB
Squeezenet	65	1.2M	914	5MB	52MB
LeNet	12	62K	230	0.4MB	2MB

Table 4.6: Torch model specification, workload characteristics, binary object size, and total enclave memory usage.

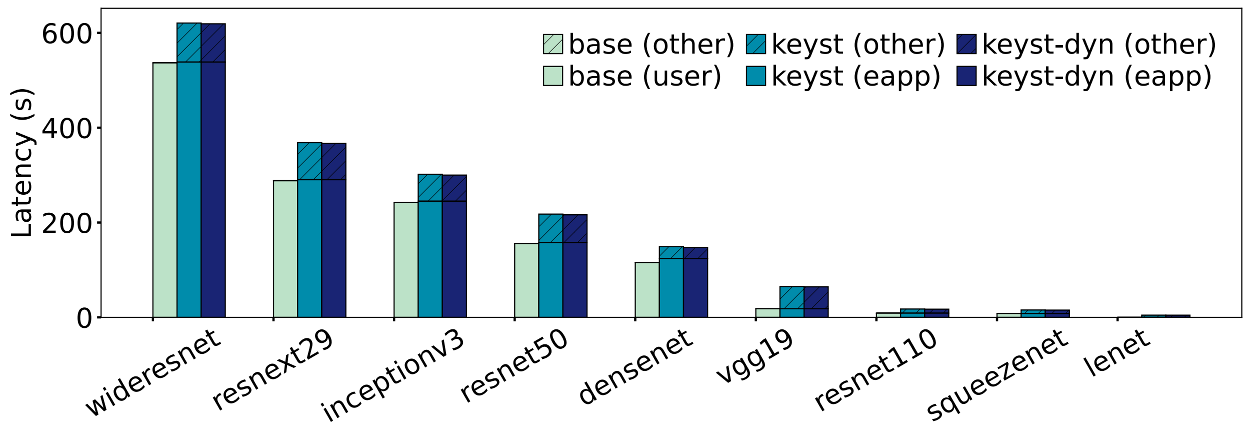

Figure 4.11: Inferencing time for various Torch models. Each bar consists of the duration of the application (`user` or `eapp`), and the other overheads (`other`). Keystone (`keyst`) and Keystone with the dynamic resizing (`keyst-dyn`) compared to native execution in (`base`).

Case Study 3: Secure Remote Computation.

The final case study implemented a secure server eapp (and remote client) to count words in an input message using the Eyrie and baseline SM. It performs attestation, uses `libsodium` to bind a secure channel to the attestation report, then polls the host for encrypted messages using edge-calls, processes them inside the enclave, and returns an encrypted reply to be sent to the client. The eapp has secure channel code (60 LoC), the edge-wrapping interface (45 LoC), and other logic (60 LoC). The host is 270 LoC and the remote client is 280 LoC. Keystone takes 45K cycles for a round-trip with an empty message, secure channel, and message passing overheads. It takes 47K cycles between the host getting a message and the enclave notifying the host to send a reply.

4.8 Related Work

This section surveys TEEs and design trade-offs that have been explored in existing works.

TEE Architectures & Extensions

Three TEEs are closely related to Keystone: (a) Intel Software Guard Extension (SGX) executes user-level code in an isolated virtual address space backed by encrypted RAM pages [126]; (b) ARM TrustZone divides the memory into two worlds (i.e., normal vs. secure) to run applications in protected memory [13]; and (c) Sanctum uses a machine-mode SM, the memory management unit (MMU), and cache partitioning to isolate enclave memory and prevent controlled-channel and cache side-channel attacks [50]. Several other TEEs explore design at layers such as hypervisors [122, 45, 82], physical memory [39, 123, 103], virtual memory [171, 51, 31], and process isolation [197, 199, 53, 157]. Interested readers can refer to Table 4.1 for a summary of TEE design choices.

Re-purposing Existing TEEs for Modularity

One way to meet Keystone's design goal of customizable TEEs is to reuse the TEE solutions that are available on commodity CPUs. For each TEE, it is possible to enable a subset of programming constructs (e.g., threading, dynamic loading of binaries) by including a software management component inside the enclave [146, 209, 24]. Alternatively, adding hardware extensions which are specifically designed and implemented for adding TEE capabilities requires lot of efforts [50, 143]. Another approach is to simulate the programmable layer, say with a trusted hypervisor layer, which then executes an untrusted OS, but potentially inflates the TCB.

Differences from Trusted Hypervisor

Keystone executes the enclave logic in the supervisor mode (RT) and the user mode (eapp), while the machine mode code (SM) merely checks and enforces isolation boundaries. Although Keystone may seem similar to a trusted hypervisor, it does not implement or perform any resource management, virtualization, or scheduling in the SM. It merely checks if the untrusted OS and the enclave (RT, eapp) are managing the shared resources correctly. Thus, Keystone SM is more analogous to a reference monitor [10, 8].

TEE Support

Several works enhance existing TEEs. At the SM layer they optimize program-critical tasks [50, 171, 20]. At the hypervisor layer they add support for multiplexing the secure isolation enforced by hardware or use nested virtualization for isolation [84, 27, 51]. At the RT layer, they target portability, functionality, security [209, 186, 15, 24, 158, 146, 144]. At the eapp layer they reduce the developer efforts [18]. Although these systems are a fixed configuration in the TEE design space, they provide valuable lessons for Keystone features and optimization.

Enhancing the Security of TEEs

Better and secure TEE design has been a long-standing goal, with advocacy for security-by-design [153, 87]. Keystone is not vulnerable to a large class of side-channel attacks [36, 220] by design, while speculative execution attacks [102, 35] are limited to out-of-order RISC-V cores (e.g., BOOM) and do not affect most SOC implementations (e.g., Rocket). Keystone can re-use known cache side-channel defenses [29, 100] as demonstrated in Section 4.4.6. Lastly, Keystone can benefit from various RISC-V proposals underway to secure IO operations with PMP [151]. Thus, Keystone either eliminates classes of attacks or allows integration with existing techniques.

Formally Verified Hardware & Software

TEE-like guarantees can be achieved orthogonally by a hardware and software stack which is formally verified as resistant against all classes of attacks that TEEs prevent. A careful and ground-up design with verified components [101, 78, 137] may provide stronger guarantees and Keystone can help explore designs which combine these with hardware protection [65, 196].

Resemblance with traditional kernel designs

Despite being designed for the TEE threat model, Keystone borrows and builds on well-known principles from a long line of work in OS design. Specifically, the choice of separating isolation (SM) and functionality (RT) has been explored mainly in micro-kernels [115]. Further, like many other works, the SM is inspired by the concept of reference monitors [10, 8]. Lastly, the modularity of abstraction between the host OS, the RT and eapp is similar to exokernels [60].

4.9 Summary

This chapter presented Keystone, the first framework for customizable TEEs. With its modular design, Keystone showcases several use cases for standard benchmarks and applications on illustrative RTs and various deployment platforms. Keystone serves as a framework for both TEE research and future deployment of novel TEE designs.

Chapter 5

Agile and Secure Implementation of New Features

As discussed in Chapter 4, an open framework can help explore various TEE threat models with reasonable performance trade-offs. However, today's workloads often require new functional capabilities and performance optimizations. What should one do when the TEE cannot meet such requirements? How should one reason about security when they make a modification to the TEE?

This chapter chooses one of the most frequently studied limitations of TEEs, *the lack of ability to share memory*. Memory sharing is essential for server programs to run efficiently in enclaves, where existing solutions either incur too much overhead or fail to provide formal reasoning. This chapter shows one way of enabling secure memory sharing by modifying the TEE itself. Specifically, it proposes to add two new native operations to the TEE platform. Then, it shows how one can formally reason about such changes using an abstract model.

5.1 Introduction

The hardware enclave [126, 110, 50, 65, 13, 6, 96] is a promising method of protecting a program [209, 158, 160, 142] by allocating a set of physical addresses accessible only from the program. The key idea of hardware enclaves is to isolate a part of physical memory by using hardware mechanisms in addition to a typical memory management unit (MMU). The isolation is based on a *disjoint memory assumption*, which constrains each of the isolated physical memory regions to be owned by a specific enclave. A hardware platform enforces the isolation by using additional in-memory metadata and hardware primitives. For example, Intel SGX maintains a per-physical-page metadata called the Enclave Page Cache Map (EPCM) entry, which contains the enclave ID of the owner [49]. The hardware looks up the entry for each memory access to ensure that the page is accessible only when the current enclave is the owner.

However, the disjoint memory assumption also significantly limits enclaves in terms of their performance and programmability. First, the enclave needs to go through an expensive initialization whenever it launches because the enclave program cannot use shared libraries in the system nor clone from an existing process [113]. Each initialization consists of copying the enclave program into the enclave memory and performing *measurements* to stamp the initial state of the program. The initialization latency proportionally increases depending on the size of the program and the initial data. Second, the programmer needs to be aware of the non-traditional assumptions about memory. For instance, system calls like `fork` or `clone` can no longer rely on efficient copy-on-write memory, causing significant performance degradation [209, 158].

A few studies have proposed platform extensions to allow memory sharing of enclaves. Yu *et al.* [226] proposes Elasticlave, which modifies the platform such that each enclave can own multiple physical memory regions that the enclave can selectively share with other enclaves. An enclave can map other enclaves' memory regions to its virtual address space by making a request, followed by the owner granting the access. Li *et al.* [113] proposes Plug-In Enclave (PIE), which is an extension of Intel SGX. PIE enables faster enclave creation by introducing a *shared enclave region*, which can be mapped to another enclave by a new SGX instruction `EMAP`. `EMAP` maps the entire virtual address space of a pre-initialized *plug-in* enclave. Although the prior work shows that memory sharing can substantially improve performance, they do not provide formal guarantees about the security of their designs.

Unsurprisingly, the disjoint memory assumption of enclaves is crucial for the security of the enclave platform and is often used to formally reason about the security of the platform. Many previous studies [196, 136, 65, 173] formally prove high-level security guarantees of enclave platforms such as non-interference properties, integrity, and confidentiality based on this assumption. However, to my best knowledge, no model formally verifies the security guarantees of an enclave platform that allows memory sharing. Memory sharing weakens this disjoint memory assumption, and as a result, necessitates formal verification under the weakened assumption.

Practical formal verification requires making choices about the right level of abstraction at which to model and apply automated reasoning. Verification on models that conform to the low-level implementation [136] or source-level code [101, 5, 46, 182, 189] is often platform-specific, in that it only provides security guarantees to those implementations and thus does not apply generally. If one seeks to verify that a memory sharing approach on top of a family of enclave platforms is secure, it is not easy to reuse verification efforts for specific implementations. This section seeks an approach that is incremental and also applicable to existing platforms.

Moreover, there are many ways one could design a memory sharing model, each varying in their complexity and flexibility. Complex models can provide more flexibility to optimize the applications for performance, but this often comes at the cost of increasing the complexity of formal verification. However, if memory sharing is too restrictive, it also becomes hard for programmers to leverage it for performance improvements. Thus, this section seeks a simple sharing model with a balance between flexibility and ease of verification.

This section presents *Cerberus*, a formal approach to secure and efficient enclave memory sharing. Cerberus chooses the *single-sharing* model (Section 5.3.1), which allows each enclave to access only one shared memory. This design decision significantly reduces the cost of verification by simplifying invariants, as well as the cost of implementation. This chapter formalizes an enclave platform model that can accurately capture high-level semantics of the extension and formally verify a property called *Secure Remote Execution* (SRE) [196]. This section performs *in-*

cremental verification by starting from an existing formal model called Trusted Abstract Platform (TAP) [196] for which the SRE property is already established. Finally, this section shows the feasibility of Cerberus by implementing it in an existing platform, RISC-V Keystone [110]. Cerberus can substantially reduce the initialization latency without incurring significant computational overhead.

To summarize, the contributions of this chapter are as follows:

- Provide a *general* formal enclave platform model with memory sharing that weakens the disjoint memory assumption and captures a family of enclave platforms
- Formally verify that the modified enclave platform model satisfies SRE property via automated formal verification
- Provide programmable interface functions that can be used with existing system calls
- Implement the extension on an existing enclave platform, and demonstrate that Cerberus reduces enclave creation latency

5.2 Formal Reasoning about TEE

5.2.1 The Secure Remote Execution Property

As mentioned earlier, much of the prior work identifies integrity and confidentiality as key security properties for enclave platforms. Thus, this work aims to prove a property that is at least as strong as these two, which is the SRE property. To provide intuition behind the property, the typical setting for an enclave user is that the user wishes to execute their enclave program securely on a remote enclave platform. The remote platform is largely untrusted, with an operating system, a set of applications, and other enclaves that may potentially be malicious. Thus, it is desirable to create a secure channel between the enclave program and user in order to set up the enclave program securely. Consequently, ensuring end-to-end security requires the enclave platform to behave in the following three ways:

- The measurement of an enclave on the remote platform can guarantee that the enclave is setup correctly and runs in a deterministic manner,
- each enclave program is integrity-protected from the untrusted entities and thus executes deterministically,
- and each enclave program is confidentiality-protected to avoid revealing secrets to the untrusted entities.

These three behaviors manifest as the secure measurement, integrity and confidentiality properties as defined in Section 5.5 and are ultimately what the platform model extended with Cerberus guarantees.

5.2.2 Formal Models of Enclave Platforms

Prior work has formally modeled and verified enclave platform models for both functional correctness and adherence to safety properties similar to the SRE property. This section discusses the approaches and explain why Cerberus chooses to extend the TAP model.

Komodo [65] is an approach for attested, on-demand and concurrent isolated enclave program execution, where the management of enclaves is delegated to a software security monitor. Their approach uses ARM's TrustZone [13] to build enclaves and aims to achieve a level of security similar to the security of Intel's SGX. They employ formal verification to verify source-level code to prove functional correctness and prove integrity and confidentiality against an adversary that controls the operating system and colludes with other enclaves. However, modeling done at the source code level means that the verification is closely tied to the implementation and makes this approach less suitable to apply to the extension.

Serval [136] on the other hand, focuses on automating the translations and verification of implementation models at the executable binary level. Serval proves properties such as the absence of undefined behavior, state-machine refinement and noninterference. The authors apply their approach to both CertikOS [78, 77] and Komodo. However, working with instruction-level models to prove high-level security properties is difficult and tedious because of a lack of program information passed to the binary level (e.g. variable names). Additionally, this chapter aims to verify the enclave platform memory sharing approach for a general model that captures the behavior of a family of enclave platforms and not a specific platform. Thus, Serval is not well suited for the goals; however, it can complement this chapter's approach by verifying that a given platform implementation binary refines the model.

Trusted Abstract Platform model [196] is an abstraction of enclave platforms that was introduced with the SRE property. The SRE property states that enclave execution on a remote platform follows its expected semantics and is confidentiality-protected from a class of adversaries defined along with the TAP model. This property provides end-to-end verification of integrity and confidentiality for enclaves running on a remote platform. It has also been formally proven that state-of-the-art enclave platforms such as Intel's SGX [126, 49] and MIT's Sanctum [50, 108] refine the TAP model and hence satisfy SRE against various adversary models.

To my best knowledge, the TAP is the only model for formal verification that has been used to capture enclave platforms in a general way. The level of abstraction also makes it readily extensible. For these reasons, this chapter's approach extends the TAP model.

5.3 Enabling Enclave Memory Sharing

Several design decisions were made for Cerberus to conform to the design goals. As alluded to earlier, the memory sharing model and interface designs are crucial for the modeling, verification, and implementation performance. This section discusses the details of how to design the memory sharing model and interface below.

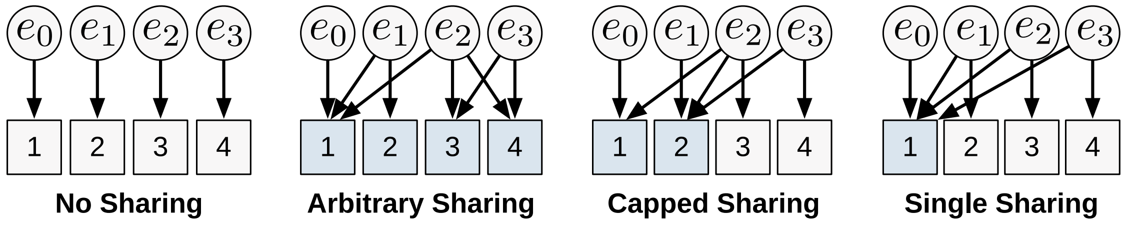

Figure 5.1: Memory sharing models with varying flexibility. Blue (and white) boxes indicate shareable (and non-shareable) physical memory region, and circles indicate enclaves. An edge from an enclave to a physical memory is an *access relation* stating that an enclave can access the memory it points to.

5.3.1 Memory Sharing Model

Figure 5.1 shows four different memory sharing models with varying levels of flexibility. This section discusses the implications for the implementation and the feasibility of formal verification for each model. This section uses the number of *access relationships* between enclaves and memory regions as a metric for the complexity of both verification and implementation.

No Sharing A *no-sharing* model refers to a model that assumes the disjoint memory assumption. This model is implemented in several state-of-the-art enclaves [126, 110, 65] and has already been previously formally verified in the TAP model [196]. The no-sharing model strictly disallows sharing memory and assigns each physical address to only one enclave. As a result, the number of access relationships is $O(max(m, n)) = O(m)$, where m is the number of physical memory regions, and n is the number of enclaves. m could be greater than n if there exists more than one physical memory regions with different properties (e.g., permissions). Thus, no-sharing implementations will require metadata scaling with $O(m)$ to maintain the access relationships. For instance, each SGX EPC page has a corresponding entry in EPCM, which contains the owner ID of the page.

Arbitrary Sharing As in Elasticlave, one could completely relax the sharing model and allow any arbitrary number of enclaves to share memory. The *arbitrary-sharing* model refers to this sharing model. In this case, the number of access relationships between enclaves is $O(mn)$. Consequently, arbitrary sharing requires metadata scaling with $O(mn)$. Since this is not scalable, the feasibility of the implementation often limits the global number of relationships. For example, Elasticlave caps m by the number of processor's PMP entries, which cannot usually exceed 64 because of hardware implementation cost [217]. Formally verifying this model using TAP can also be complex due to the flexibility of the model. Verifying security properties such as SRE requires reasoning about safety properties with multiple traces and platform invariants with nested quantifiers. Modeling an arbitrary number of shared enclave memory would add to this complexity. For example, one inductive invariant needed to prove SRE on TAP is the invariant that memory accessible by a given enclave is owned by itself. If the platform allows an arbitrary number of enclaves to be accessible by a given enclave, one encoding for the extension of this invariant existentially quantifies over all of the enclaves to state that the owner of the memory accessible to the given enclave is one of the enclaves that owns the shared memory region (Section 5.5, Eq. (5.8)). The encoding of this invariant in TAP itself uses first-order logic with the theory of arrays and, in general, is not decidable [28]. As a result, the introduction of this quantifier further complicates the invariant. This is discussed in more detail in Section 5.5.

Capped Sharing To achieve scalability in the number of access relationships, one could constrain the sharing rule such that each enclave can only access a limited number of shared physical memory regions. A *capped-sharing* model refers to this sharing model. In Figure 5.1, capped sharing shows an example where each enclave is only allowed to access at most two additional shared physical memory regions. As an example, PIE [113] introduces a new type of enclave called *plug-in* enclave, which can be mapped to the virtual address space of a normal enclave. PIE can improve the performance of dynamically linked programs by having each shareable enclave contain a shared library, which can be mapped to enclaves using them. This reduces the number of relationships to $O(kn + m)$, where k is the number of shared physical memory regions that are allowed to be accessed by an enclave. Despite the limiting constraint in *capped sharing*, a formal model capturing any arbitrary limit k would still require modeling an arbitrary number of the shared enclave memory as in the *arbitrary sharing* scheme. As a result, formally verifying this design requires the same complicated invariant as the arbitrary-sharing model mentioned previously.

Single Sharing The *single-sharing* model only allows an enclave to access the shared memory regions of a particular enclave. It is a special case of the capped sharing with $k = 1$. Single sharing reduces the complexity to $O(m)$, the least of all sharing models. Although the figure depicts a single shared physical memory region, there can be multiple shared physical memory regions for single sharing. The single-sharing model significantly reduces the efforts of formal reasoning and implementation. First, the formal reasoning no longer requires the complex invariant mentioned in the capped and arbitrary sharing models because memory that can be accessed by an enclave either belongs to the enclave itself or only one other enclave that is sharing memory. Second, the implementation becomes much simpler as it requires only one per-enclave metadata to store the reference to the shared memory. The platform modification becomes also minimal as it checks one more metadata per memory access. Despite its simplicity, the single-sharing model can still improve the performance of programs by having all of the shared contents (e.g., shared library, initial code, and initial data) in the shareable enclave. For these reasons, Cerberus chooses the single-sharing model.

5.3.2 Interface

Separate from the sharing model, it is important to design an interface of functions such that the memory sharing can be used by the enclave program. This section describes the new operations added to the enclave platform for memory sharing.

Elasticlave and PIE introduce explicit operations to *map* or *unmap* the shareable physical memory region to the virtual address space of the enclave. For example, in order for an Elasticlave memory region to be shared, an enclave program needs to explicitly call `map` operation to request access on the region, which will be approved by the owner via `share` operation. Similarly, PIE allows an enclave to use `EMAP` and `EUNMAP` instructions to map and unmap an entire plug-in enclave memory to the virtual address of the enclave.

Both approaches allow an enclave program to map shareable physical memory regions to its virtual address space. However, there are a few downsides to the approaches. First, the programmers must manually specify which part of the application should be made shareable. The shareable regions may include text segments, static data segments, and dynamic objects (e.g., a machine learning model). In most cases, the programmers must completely rewrite a program such that the shareable part of the program is partitioned into a separate enclave memory. Second, a dynamic map or unmap requires local attestation, which verifies that the newly mapped memory is in an expected initial state. Thus, the measurement property of a program relies on the measurement property of multiple physical memory regions.

Cerberus takes an approach similar to a traditional optimization technique, which is cloning an address space with copy-on-write, as in system calls like `clone` and `fork`. In general, programmers expect such system calls to copy the entire virtual address space of a process – no matter what it contains – to a newly-created process. Using a similar approach will allow the programmers to write enclave programs with the same expectation. Also, such an interface will not require additional properties or assumptions on measurements of multiple enclaves. Since the initial code of an enclave already contains when to share its entire address space, the initial measurement implicitly includes all memory contents to be shared.

To this end, Cerberus introduces two enclave operations, which are `Snapshot` and `Clone`. `Snapshot` freezes the entire memory state of an enclave, and `Clone` creates a logical duplication of an enclave. `Snapshot` is only callable from the enclave itself, allowing the enclave to decide when to share its memory. The adversary can call `Clone` any time, which does not break the security (See Section 5.4.5). When the adversary calls `Clone` on an existing enclave, a new enclave is created and resumes with a copy-on-write (CoW) memory of the snapshot. Thus, any changes to each of the enclaves after the `Clone` are not visible to each other. The following sections formally discuss the sharing model and the interface of Cerberus, and present security arguments on the extension in detail.

Symbol	**Description**
$\mathbb{N}$	The usual set of natural numbers $\{1, 2, 3, ...\}$.
Bool	The set of Boolean values $\{$*true*,*false*$\}$.
$\in$	*Element of* operator in set theory.
$\subset$	Subset operator in set theory.
$\doteq$	*By definition* symbol. This is not to be confused with the logical equal operator $=$.
$=$	Logical equal operator.
$\iff$	If and only if operator.
$\perp$	Bottom value. A default value that is only equal to itself.
$\mathcal{OS}$	The identifier for the enclave.
e_{inv}	The invalid enclave ID. A variable assigned with this ID can be viewed as a null value.
VA	Set of virtual addresses.
PA	Set of physical addresses.
ACL	Permissions for virtual addresses (read, write and execute permissions).
$\mathcal{E}_{id}$	Set of enclave IDs $e_1, e_2, ...$, including the operating system ID $\mathcal{OS}$.
$\mathcal{E}_M$	Enclave metadata type.
$\forall x \in X.expr(x)$	Forall quantified expression; states for all $x \in X$, $expr(x)$ must be true.
$\lambda x.expr$	Function with argument x; computes $expr$.
ITE$(c, expr_1, expr_2)$	If-then-else operator that evaluates $expr_1$ if c is true or otherwise evaluates $expr_2$.
init$(E_e(\sigma))$	Enclave e in state σ has been initialized by `Launch` and has not yet been executed using `Enter`.
sufficient_mem$(\sigma.o)$	A function that returns whether there is enough memory to allocate given the memory ownership map.
valid(e_{id})	Returns whether the enclave ID e_{id} is a valid enclave ID. In other words, not equal to $\mathcal{OS}$ nor e_{inv}.
mapped$_e(v)$	Returns whether a virtual address $v \in VA$ is mapped for the enclave e.
curr(σ)	The ID of the process currently executing in state σ.
S	Set of state of TAP$_C$.
I	Set of initial states of TAP$_C$.
$\rightsquigarrow$	Transition relation of TAP$_C$.
$E_e(\sigma)$	The projection of state σ to the enclave state e.
$A_e(\sigma)$	The projection of state σ to state that is writable to the adversary.
$I_e(\sigma)$	Inputs of an enclave at state σ.
$O_e(\sigma)$	Outputs of an enclave at state σ.
π^i	The ith state of the platform trace π.
π_j	Platform trace j; not to be confused with the jth state of the trace.

Table 5.1: Glossary of Symbols used for Chapter 5

5.4 Formal Model

This section first introduce a threat model in Section 5.4.1 that is consistent with these goals and the current state-of-the-art enclave threat models. Section 5.4.2 lists and justify the assumptions, Section 5.4.3 introduces the formal models for the platform and adversary based on these assumptions, and Section 5.4.4 introduces the two new operations `Snapshot` and `Clone` of Cerberus. Section 5.4.5 concludes with high-level arguments to justify that the addition of `Snapshot` and `Clone` does not weaken the security guarantees. Figure 5.1 is a glossary of symbols used in the rest of the chapter.

Section 5.5 uses these formal models to formally verify the SRE [196] property, which is a critical security property used to prove that enclaves executing in the remote platform are running as expected and confidentially.

To recap, this section's formal models extend the TAP model introduced by Subramanyan *et al.* [196]. While SRE has been proven on the TAP model, the original TAP model makes a number of assumptions that are weakened in Cerberus design. For example, Cerberus allows memory sharing and hence the disjoint memory assumption in the TAP model is weakened.

Additionally, it is not immediately clear that the addition of these two operations clearly preserves SRE. Thus, it is needed to prove that SRE still holds under Cerberus-extended model of TAP. The rest of the literature refers to the original formal platform model defined by Subramanyan *et al.* [196] as the base TAP model and Cerberus-extended model as TAP$_C$.

5.4.1 Threat Model

Cerberus extension follows the typical enclave threat model where the user's enclave program e is integrity- and confidentiality-protected over the enclave states (e.g. register values and data memory owned by the enclave program) against any software adversary running in the remote enclave platform. The software adversaries of an enclave include the untrusted operating system, user programs, and the other enclaves as shown in Figure 5.2.

With Cerberus, enclaves may share data or code that were common between enclaves before the introduction of the `Clone`. Cerberus assumes that the memory is implicitly not confidential among these enclaves with shared memory. However, each enclave's memory should not be observable by the operating system or other enclaves and applications. Cerberus ensures that the enclaves are still *write-isolated*, which means that any modification to the data from one enclave must not be observable to the other enclaves, even to the enclave that it cloned from. Thus, any secret data needs to be provisioned after the enclave is cloned. It is the enclave programmer's responsibility to make sure that the parent enclave does not contain any secret data that can be leaked through the children.

Cerberus does not consider the program running in the enclave to be vulnerable or malicious by itself. For example, a program can generate a secret key in the shared memory, and encrypt the confidential data of the child with the key. This would break confidentiality among children enclaves write-isolated from each other because the children will have access to the key in the

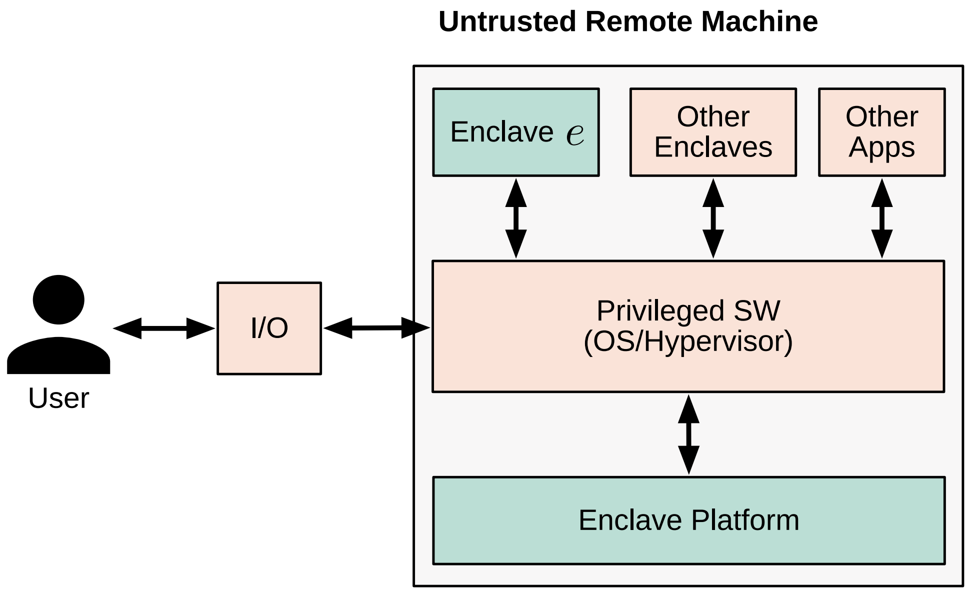

Figure 5.2: A user provisions their (protected) enclave e in the remote enclave platform isolated from untrusted software. Green/red boxes indicate trusted/untrusted components.

shared memory. This could be easily solved by having programs load secrets to their memory after they have created the distrusting children.

In some platforms such as Intel® SGX, side-channel attacks are out-of-scope in their threat model. Since the main goal is to design a generic extension, Cerberus also does not consider any type of side-channel attack or architecture-specific attack [124, 37, 102, 117, 35, 212, 176, 220, 132, 215, 180, 109] in this section. Since the base TAP model has also been used to prove side-channel resiliency on some enclave platforms [110, 50], it is not impossible to extend the proofs to such adversary models. Denial of service against the enclave is also out of scope of Cerberus; this is consistent with the threat models for existing state-of-the-art enclave platforms.

A formal model of the threat model is described in more detail in Section 5.4.3 after the formal definition of the platform.

5.4.2 Assumptions

This section summarizes the list of assumptions about the execution model for simplifying and abstracting the modeling below, including the assumptions mentioned above:

- Every enclave operation is treated as an atomic operation, consistent with previous work [196]. If an enclave operation returns with an error code, the states of the platform are entirely reverted to the state prior to the execution of that operation.

- State continuity of enclaves is out-of-scope of this chapter, consistent with TAP [196], and can be addressed using alternative methods [152, 92].
- The memory allocation algorithm (e.g. for copy-on-write) is deterministic given that the set of unallocated memory is the same. This means that given any two execution sequences of a platform, as long as the page table states are the same, the allocation algorithm will return the same free memory location to allocate.
- Side-channels are out of scope which is consistent with existing enclave platform adversary models as mentioned before.
- Enclave programs that wants integrity and confidentiality protection are assumed to be bug-free and do not inadvertently leak secret information through its outputs.

Next section introduces the formal model describing the platform which extends the existing TAP model with `Snapshot` and `Clone` under these assumptions.

5.4.3 Formal TAP$_C$ Platform Model Overview

As mentioned, a user of an *enclave platform* typically has a program and data that they would like to run securely in a remote server, isolated from all other processes as shown in Figure 5.2. Such a program can be run as an enclave e. The remote server provides isolation using its hardware primitives and software for managing the enclaves, where the software component is typically firmware or a security monitor. This software component provides an interface for the enclave user through a set of operations, denoted by $\mathcal{O}$, for managing e. The goal is to guarantee that this enclave e is protected from all other processes on the platform and running as expected. For the purpose of understanding the proofs, the *protected enclave* e refers to the enclave that SRE property would like to protect. This distinction differentiates e from the *adversary-controlled* enclaves.

Platform and Enclave State

The platform can be viewed as a transition system $M = \langle S, I, \rightsquigarrow \rangle$ that is always in some state denoted by $\sigma \in S$[1]. Alternatively, σ can be viewed as an assignment to a set of state variables V. The platform starts in an initial state in the set I and it transitions between states defined by a transition relation $\rightsquigarrow \subset S \times S$, where $\subset$ denotes subset and $S \times S$ is the Cartesian product of states. $(\sigma, \sigma') \in \rightsquigarrow$ means that the platform can transition from σ to σ'. An execution of the platform therefore emits a sequence of states $\pi = \langle \sigma^0, \sigma^1, ...\sigma^n \rangle$, where $(\sigma^i, \sigma^{i+1}) \in \rightsquigarrow$ for $i \in \{0, ..., n-1\}$. One can write $\pi^i = \sigma^i$ interchangeably, but will usually write π^i whenever referencing a specific trace. When an enclave is initially launched, it is in the initial state prior to enclave execution, which is indicated by the predicate $init_e(\sigma) : S \rightarrow Bool$. Next section describes the set of variables V and enclave state $E_e(\sigma)$ for TAP$_C$.

[1] $\in$ is the usual *element of* operator in set theory which states the right hand side value σ is an element of the set S.

TAP$_C$ State Variables

Each of the variables in TAP$_C$ are shown in Table 5.2. $pc : VA$[2] is an abstraction of the program counter whose value is a virtual address from the set of virtual addresses VA. $\Delta_{rf} : \mathbb{N} \rightarrow W$ is a register file that is a map [3] from the set of register indices (of natural numbers) $\mathbb{N}$ to the set of words W. $\Pi : PA \rightarrow W$ is an abstraction of memory that maps the set of physical addresses PA to a set of words. One writes $\Pi[a]$ to represent the memory value at a given physical address $a \in PA$. A page table abstraction defines the mapping of virtual to physical addresses a_{PA} and access permissions $a_{perm} : VA \rightarrow ACL$, where ACL is the set of read, write, and execute permissions. ACL can be defined as the product $VA \rightarrow Bool \times Bool \times Bool$, where $Bool \doteq \{true, false\}$ and the value of the map corresponds to the read, write, and execute permissions for a given virtual address index. $e_{curr} : \mathcal{E}_{id}$ represents the current enclave that is executing. $\mathcal{E}_{id} = \mathbb{N} \cup \{\mathcal{OS}\} \cup \{e_{inv}\}$[4] is the set of enclave IDs represented by natural numbers and a special identifier $\mathcal{OS}$ representing the untrusted operating system. The identifier e_{inv} is reserved to refer to the invalid enclave ID which can be thought of as a default value that does not refer to any valid enclave. For the ease of referring to whether an enclave is valid and launched, one can use the predicate $valid(e_{id}) \doteq e_{id} \neq e_{inv} \wedge e_{id} \neq \mathcal{OS}$ that returns whether or not an ID is a valid enclave ID. $\wedge$ refers to the usual logical and operator. o is a map that describes the ownership of physical addresses, each of which can be owned by an enclave (with the corresponding enclave ID) or the untrusted operating system.

Lastly, each enclave e has a set of enclave metadata $\mathcal{M}$, which is a record of variables described in Table 5.3. One can abuse notation and write $\mathcal{M}^{pc}[e]$ to represent the program counter $\mathcal{M}^{pc}$ of e in the record stored in the metadata map $\mathcal{M}$. The notation $[\cdot]$ is similarly used for other metadata fields defined in Table 5.3. $\mathcal{M}^{EP}[e]$ is the entry point of the enclave that the enclave e starts in after the `Launch` and before `Enter`. $\mathcal{M}^{AM}_{PA}[e]$ is virtual address map of the enclave program. $\mathcal{M}^{AM}_{perm}[e]$ is map of address permissions for each virtual address. $\mathcal{M}^{EV}[e]$ is the map from virtual addresses to Boolean values representing whether an address is allocated to the enclave. $\mathcal{M}^{pc}[e]$ is the current program counter of the enclave. $\mathcal{M}^{regs}[e]$ is the saved register file of the enclave. $\mathcal{M}^{paused}$ is a Boolean representing whether or not the enclave has been paused and is initially false at launch.

These variables were introduced in the base TAP model and are unmodified in TAP$_C$. Section 5.4.4 introduces the remaining four metadata variables required for Cerberus, which are additional state variables in TAP$_C$ that are not defined in the base TAP model.

The state $E_e(\sigma)$ is a projection of the platform state to the enclave state of e that includes $\mathcal{M}^{pc}[e], \mathcal{M}^{regs}[e]$, and the projection of enclave memory $\lambda v \in VA.\textit{ITE}(\mathcal{M}^{EV}[e][v], \Pi[\mathcal{M}^{AM}_{PA}[v]], \perp)$. In the previous expression, $\lambda v \in VA.E$ is the usual lambda operator over the set of virtual addresses v and expression body E, $\textit{ITE}(c, expr_1, expr_2)$ is the if then else operator that returns $expr_1$ if condition c is true and $expr_2$ otherwise. $\perp$ is the constant bottom value which can be thought of as a don't-care or unobservable value. This projection of memory represents all memory accessible to enclave e, including shared memory and memory owned by enclave e as referenced by the virtual address map $\mathcal{M}^{AM}_{PA}$.

[2]Write v: T to mean variable $v \in V$ has type T

[3]of type $L \rightarrow R$, where the index type is L and value type is R.

[4]$\{\cdot\}$ is the singleton set

State Var.	**Type**	Description
pc	VA	The program counter.
Δ_{rf}	$\mathbb{N} \rightarrow W$	General purpose registers.
Π	$PA \rightarrow W$	Physical memory.
a_{PA}	$VA \rightarrow PA$	Page table abstraction; virtual to physical address map.
a_{perm}	$VA \rightarrow ACL$	Page table abstraction; virtual to their permissions.
e_{curr}	$\mathcal{E}_{id}$	Current enclave. $e = \mathcal{OS}$ means the OS is running.
o	$PA \rightarrow \mathcal{E}_{id}$	Map from physical addresses to the enclave that owns it.
$\mathcal{M}$	$\mathcal{E}_{id} \rightarrow \mathcal{E}_M$	Map of enclave IDs to enclave metadata. $\texttt{emd}[\mathcal{OS}]$ stores a checkpoint of the OS.

Table 5.2: Description of TAP State Variables

State Var.	**Type**	**Description of each field**
$\mathcal{M}^{EP}$	VA	Enclave entrypoint.
$\mathcal{M}^{AM}_{PA}$	$VA \rightarrow PA$	Enclave's virtual address map.
$\mathcal{M}^{AM}_{perm}$	$VA \rightarrow ACL$	Enclave's address permissions.
$\mathcal{M}^{EV}$	$VA \rightarrow Bool$	Set of private virtual addresses.
$\mathcal{M}^{pc}$	VA	Saved program counter.
$\mathcal{M}^{regs}$	$\mathbb{N} \rightarrow W$	Saved registers.
$\mathcal{M}^{paused}$	$Bool$	Whether enclave is paused.
$\mathcal{M}^{IS\dagger}$	$Bool$	Whether the enclave is a snapshot.
$\mathcal{M}^{CC\dagger}$	$\mathbb{N}$	Number of children enclaves.
$\mathcal{M}^{RS\dagger}$	$\mathcal{E}_{id}$	Enclave's root snapshot.
$\mathcal{M}^{PAF\dagger}$	$PA \rightarrow Bool$	Map of free physical addresses.

Table 5.3: Description of TAP $\mathcal{E}_M$ enclave metadata record

$$\begin{tikzcd}
\pi_1^0 \arrow[r, "op^0"] \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \cdots & \pi_1^i \arrow[r, "\mathcal{A}"] \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \pi_1^{i+1} \arrow[r, "op^{i+1}"] \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \pi_1^{i+2} \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \cdots \\
\pi_2^0 \arrow[r, "op^0"] & \cdots & \pi_2^i \arrow[r, "\mathcal{A}"] & \pi_2^{i+1} \arrow[r, "op^{i+1}"] & \pi_2^{i+2} & \cdots
\end{tikzcd}$$

Figure 5.3: Illustrating the execution of two traces of the platform in the secure measurement, integrity and confidentiality proofs. Proof obligations for each property are checked as indicated by $\approx_{\mathcal{L}}$ and equal initial condition indicated as $\approx_{\mathcal{L}}$. op^i indicates enclave execution of an operation from $\mathcal{O}$ at step i and $\mathcal{A}$ indicates an adversary execution.

Enclave Inputs and Outputs

Communication between an enclave e and external processes for a given state σ are controlled through e's inputs $I_e(\sigma)$ and its outputs $O_e(\sigma)$. $I_e(\sigma)$ includes the arguments to the operations that manage enclave e, areas of memory outside of the enclave that the enclave may access and an untrusted attacker may write to, and randomness from the platform. $O_e(\sigma)$ contains the outputs of enclave e that are writable to by e and accessible to the attacker and the user.

Platform and Enclave Execution

An execution of an enclave e is defined by the set of operations from $\mathcal{O}$, in which the execution of an operation is deterministic up to its input $I_e(\sigma)$ and current state $E_e(\sigma)$. This means that given the same inputs $I_e(\sigma)$ and enclave state $E_e(\sigma)$, the changes to enclave state $E_e(\sigma)$ is deterministic. The set of operations for the base TAP model is $\mathcal{O}_{base} \doteq \{\texttt{Launch}, \texttt{Destroy}, \texttt{Enter}, \texttt{Exit}, \texttt{Pause}, \texttt{Resume}\}$. TAP$_C$ extends the base set with two additional operations: $\mathcal{O} \doteq \mathcal{O}_{base} \cup \{\texttt{Snapshot}, \texttt{Clone}\}$[5]. One can use the predicate $curr(\sigma) = e$ to indicate that enclave e is executing at state σ and $curr(\sigma) = \mathcal{OS}$ to indicate that the operating system is executing [6].

Formal Adversary Model

In the model, untrusted entities such as the OS and untrusted enclaves are represented by an adversary $\mathcal{A}$ that can make arbitrary modifications to state outside of the protected enclave e, denoted by $A_e(\sigma)$. Consistent with the base TAP model, the untrusted entities and protected enclave e takes turns executing under interleaving semantics in the formal TAP$_C$ model, as illustrated in Figure 5.3.

Conventionally, an adversary can be defined with an observation and tamper function that describes what the adversary can observe and change in the platform state during its execution to break integrity and confidentiality. Below describes these two functions.

[5] $\cup$ is the union operation over sets

[6] The adversary can execute the operating system or one of its controlled enclaves.

Tamper Function The tamper function is used to model these malicious modifications to platform state by the adversary and is defined over $A_e(\sigma)$ which includes any memory location that is not owned by the protected enclave e and page table mappings. The semantics of the model allows the adversary to make these changes whenever it is executing. The model allows all tampered state to be unconstrained, which means they can take on any value. This type of adversary tamper function over-approximates what the threat model can change and is typically referred to as a havocing adversary [196, 40].

Observation Function The adversary's observation function is denoted $obs_e(\sigma)$. The model allows the adversary to observe locations of memory that are not owned by the protected enclave e, described by the set $obs_e(\sigma) \doteq O_e(\sigma) \doteq \lambda p \in PA.ITE(\sigma.o[p] \neq e, \sigma.\Pi[p], \perp)$. Intuitively, obs_e is a projection of platform state that is observable by the adversary whose differences should be excluded by the property. For example, if the same enclave program operating over different secrets reveals secrets through the output, that is a bug in the enclave program, which disagrees with the TAP assumptions. The adversaries may try to modify or read the enclave state during the lifetime of the enclave.

Under this threat model, the following section proves that the TAP$_C$ model still satisfies the SRE property.

5.4.4 The Extended Enclave Operations

Cerberus is the extension of enclave platforms with two new operations `Snapshot` and `Clone` to facilitate memory sharing among enclaves. Intuitively, `Snapshot` converts the enclave executing the operation into a read-only enclave and `Clone` creates a child enclave from the parent enclave being cloned so that the child enclave can read and execute the same memory contents as the parent at the time of clone.

This extension requires four new metadata state variables that are indicated in Table 5.4.3 with the † symbol. $\mathcal{M}^{IS}[e]$ is a Boolean valued variable indicating whether or not `Snapshot` has been called on the enclave e. $\mathcal{M}^{CC}[e]$ is the number of children e has, or in other words, the number of times clone has been called on the enclave e where e is the parent of `Clone`. $\mathcal{M}^{RS}[e]$ is a reference to the root snapshot of e if one exists, and $\mathcal{M}^{PAF}[e]$ is a map of addresses that have been assigned to e but are not yet allocated memory.

This section defines the semantics of the two new operations introduced in Cerberus.

`Clone`

`Clone` creates a clone of an existing logical enclave such that there exists two enclaves with identical enclave states. `Clone` alone provides a functionality similar to `fork` and `clone` system calls, no matter whether the platform enables memory sharing. More concretely, the `Clone` takes in three arguments: the ID of the existing parent enclave $e^p_{id} \in \mathcal{E}_{id}$ to clone, the enclave ID of the child enclave $e^c_{id} \in \mathcal{E}_{id}$ and a set of physical addresses assigned to the enclave $x_p \subset PA$. The assigned physical addresses are marked as free (i.e., $\mathcal{M}^{PAF}[e_c][p] = true, \forall p \in x_p$), so that they

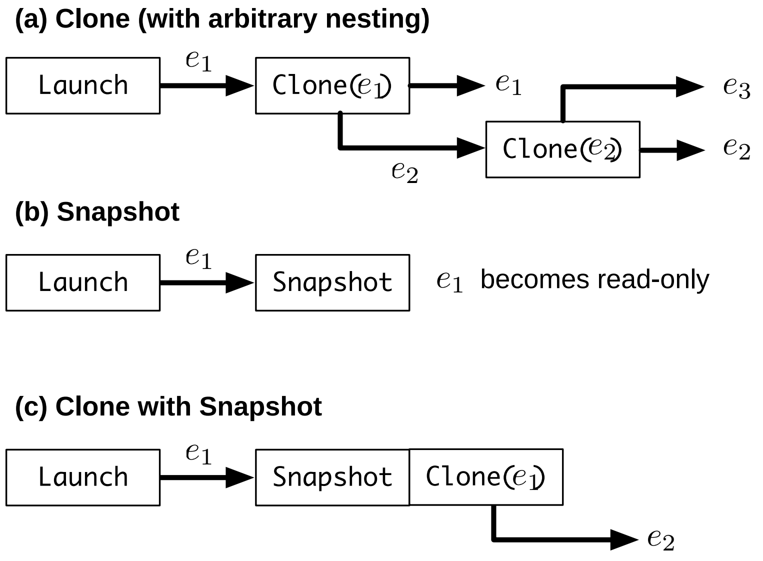

Figure 5.4: `Clone`, `Snapshot`, and `Clone` with `Snapshot`.

can be used for copying parent's memory. The child enclave e_c with corresponding enclave ID e^c_{id} is used to create a clone of the parent e_p such that $E_{e_p}(\sigma) = E_{e_c}(\sigma)$. In other words, the virtual memory of both enclaves are equal. $E_{e_p}(\sigma_0)$ denotes the initial state of the parent such that $init(E_{e_p}(\sigma))$.

`Clone` is a special way of creating an enclave; instead of starting from the initial enclave state $E_{e_p}(\sigma_0)$, `Clone` allows an enclave to start from an existing enclave e_p, which is effectively identical to creating two enclaves with the same initial state and then executing the same sequence of inputs up until the point clone was called, as explained in Section 5.4.5.

To prevent the malicious use of clone, `Clone` requires the condition Eq. 5.1 to hold when it is called with state σ.

$$\begin{aligned}
&valid(\sigma.e_{curr}) && \wedge \\
&e^c_{id} \neq e^p_{id} && \wedge \\
&e^c_{id} \neq e_{inv} \wedge e^p_{id} \neq e_{inv} && \wedge \\
&\forall p \in PA.p \in x_p \Rightarrow \sigma.o[p] = \mathcal{OS} && \wedge \\
&\mathit{sufficient\_mem}(\sigma.o)
\end{aligned} \tag{5.1}$$

This condition states that the `Clone` succeeds if and only if both the parent and child enclave IDs are valid, both the parent and child enclave IDs do not reference each other, all physical addresses in x_p are owned by the OS (and thus can be allocated to the enclave), and there is *sufficient memory* to be allocated to the enclave.

If clone is called successfully, `Clone` copies all of the data in the virtual address space of e_p to e_c to ensure write-isolation. For each virtual address v mapped by e_p (mapped), `Clone` first selects a physical address p owned by e_c, copies the contents from $\Pi[\mathcal{M}_{PA}^{AM}[e_p][v]]$ to $\Pi[p]$, and update the page table of e_c such that $\mathcal{M}_{PA}^{AM}[e_c][v] = p$. This can be implemented in the platform itself (i.e., the security monitor firmware in Keystone) or in a local vendor-provided enclave (i.e., similar to the Quoting Enclave in Intel® SGX).

In Eq. 5.1, *sufficient_mem* $: PA \rightarrow \mathcal{E}_{id} \rightarrow Bool$, *sufficient_mem* can be viewed as a predicate that determines whether there is enough memory to copy all data. *sufficient_mem* is modeled abstractly in the TAP$_C$ model to avoid an expensive computation to figure out whether there is enough memory.

`Clone` is only called from the untrusted OS, because it requires the OS to allocate resources for the new enclave. Thus, if an enclave program needs to clone itself, it needs to collaborate with the OS to have it call `Clone` on behalf. As the newly-created enclave is still an isolated enclave, the SRE property on both parent and child enclaves should hold even with a malicious OS.

`Snapshot`

`Clone` by itself still requires copying the entire virtual memory to ensure isolation. To enable memory sharing, `Snapshot` makes the caller enclave e to be an immutable image (Figure 5.4b). After calling `Snapshot`, e becomes a special type of enclave referred to as a *snapshot enclave* or the *root snapshot* of its descendants. e is no longer allowed to execute at this point because all of its memory becomes read- or execute-only. On the other hand, e can be *cloned* by `Clone`, where the descendants of e are allowed to read directly from the e's shared data pages. Any writes from the descendants to physical addresses $p \in PA$ owned by e (i.e., $\sigma.o[p] = e$) trigger copy-on-write (CoW). This scheme ensures that the descendant enclaves are still write-isolated from each other.

Like `Clone`, `Snapshot` has a success condition described in Eq. (5.2). The condition checks that the current executing enclave is valid $valid(\sigma.e_{curr})$, e is not ($\neg$) already a snapshot and the enclave cannot have a root snapshot (which is described in the next section) in the current state σ.

$$valid(\sigma.e_{curr}) \wedge \neg\sigma.\mathcal{M}^{IS}[e] \wedge \neg valid(\sigma.\mathcal{M}^{RS}[\sigma.e_{curr}]) \tag{5.2}$$

If `Snapshot` is called successfully in a state that satisfies this condition, e is marked as a snapshot enclave. In the formal model, the metadata state $\mathcal{M}^{IS}[e]$ is to *true*.

`Clone` after `Snapshot`

In order to make `Clone` work with `Snapshot`, `Clone` additionally increments e_p's child count $\mathcal{M}^{CC}[e_p]$ by 1, and sets the root snapshot of e_c (i.e., $\mathcal{M}^{RS}[e_c]$) to either e_p or e_p's root snapshot $\mathcal{M}^{RS}[e_p]$ if it has one.

With the single-sharing model, arbitrarily nested calls of `Clone` should still keep only one shareable enclave. As shown in Figure 5.5, there will be only one root snapshot e_1, whose memory is shared across all the descendants. This means that even though cloning can be arbitrarily nested, the maximum height of the tree representing the root snapshot to child enclave is one.

To maintain the same functionality, the virtual address space of the parent and the child should be the same right after `Clone`. Thus, a descendant enclave memory will diverge from the shared

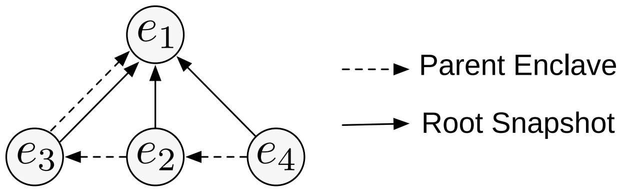

Figure 5.5: Parent-child relationship and root snapshot-child relationship of four enclaves in Cerberus. Enclave e_1 is a snapshot and parent enclave of e_2, which is the parent of e_3, which is the parent of e_4. Despite the nested parent relationship, the root snapshot of e_2, e_3, and e_4 are e_1.

memory when the descendant writes. Unfortunately, there is no better way than having `Clone` copy the diverged memory from the parent to the child. This is a limitation of Cerberus because the benefit of sharing memory will gradually vanish as the memory of descendant diverges from the snapshot. However, Cerberus is very effective when the enclaves mostly write to a small part of memory while sharing the rest. It is the programmer's responsibility to optimize their program by choosing the correct place to call `Snapshot`.

5.4.5 Security Arguments

Security Arguments for Clone Allowing the untrusted OS to clone the enclave any number of times does not break the security guarantees of the enclave. This becomes more obvious by viewing cloning as a special way of creating an enclave. No matter how many enclaves the untrusted adversary creates using `Clone`, the final machine state $E_e(\sigma_n)$ after the operation will look as if the OS launched multiple enclaves e_1 and e_2 with the same initial state $E_{e_1}(\sigma_0) = E_{e_2}(\sigma_0)$ and executed e_1 and e_2 with the same enclave input sequence for both enclaves such that $I_{e_1}(\sigma_0) = I_{e_2}(\sigma_0), I_{e_1}(\sigma_1) = I_{e_2}(\sigma_1), ..., I_{e_1}(\sigma_n) = I_{e_2}(\sigma_n)$ to reach the state $E_{e_1}(\sigma_n) = E_{e_2}(\sigma_n)$.

The OS is allowed to refuse to clone the enclave upon the enclave's request, but again, this is denial of service and is out of scope in the enclave threat model.

There is no uniqueness guarantee with the addition of `Clone`; for example, the measurement verifier cannot distinguish two different enclaves with the same nonce. However, this is already not a property that an enclave provides, and so does not represent the loss of security.

Security Arguments for Snapshot `Snapshot` is an irreversible operation that each enclave can call only once. `Snapshot` can be called only by the enclave itself. Thus, the initial enclave code needs to contain the code that calls `Snapshot`. Since the enclave decides when it wants to freeze, the snapshot enclave is inheritably trusted if the enclave's initial state is trusted.

5.5 Proving Formal Security Guarantees

To recap, one of this chapter's goals is to prove that Cerberus extension applied to an enclave platform does not weaken the high-level security property SRE. This is accomplished by reproving SRE on TAP$_C$. As per Theorem 3.2 [196], it suffices to show that the triad of properties secure measurement, integrity and confidentiality hold on TAP$_C$ to prove SRE. This section formally defines each of these properties along with brief justification as to why these properties hold in the TAP$_C$ model against the adversary described in Section 5.4. Each of these properties have been mechanically[7] proven on the base TAP model [196] without `Snapshot` and `Clone`, and this work extends these proofs to provide the same guarantees[8] on the extended TAP$_C$ model. For brevity, this section leaves out some of the model implementation details. This section also provides a list of additional inductive invariants required to prove the properties in the TAP$_C$ model at the end of this section.

5.5.1 Secure Remote Execution

Secure Measurement In any enclave platform, the user desires to know that the enclave program running remotely is in fact the program that it intends to run. In other words, the platform must be able to *measure* the enclave program to allow the user to detect any changes to the program prior to execution. Intuitively, the first part of the measurement property is formalized as Eq. (5.3) stating that if any two enclaves e_1 and e_2 are in their initial states, the measurements of each enclave $\mu(e_1)$ and $\mu(e_2)$ are the same after calling `Launch` if and only if the enclaves must have identical initial states. μ is defined to be the measurement function that the user would use to check that their enclave e is untampered with in the remote platform.

$$\begin{aligned} &\forall \sigma_1, \sigma_2 \in S. \big(init(E_{e_1}(\sigma_1)) \wedge init(E_{e_2}(\sigma_2)) \big) \quad \Rightarrow \\ &\qquad \big(\mu(e_1) = \mu(e_2) \iff E_{e_1}(\sigma_1) = E_{e_2}(\sigma_2) \big) \end{aligned} \tag{5.3}$$

The second part of measurement ensures that the enclave executes deterministically given an initial state. This is formalized as Eq. (5.4), which states that any two enclaves e_1 and e_2 starting with the same initial states, executing in lock step and with the same inputs at each step, should have equal enclave states and outputs throughout the execution.

[7]these mechanical proofs were written in the automated verification toolkit UCLID5 [181, 210] which uses an SMT solver in the backend.

[8]Same guarantees for the memory adversary.

$$\begin{aligned}
&\forall \pi_1, \pi_2. && \quad (5.4)\\
&\big(E_{e_1}(\pi_1^0) = E_{e_2}(\pi_2^0) && \wedge\\
&\forall i.(curr(\pi_1^i) = e_1) \iff (curr(\pi_2^i) = e_2) && \wedge\\
&\forall i.(curr(\pi_1^i) = e_1) \Rightarrow I_{e_1}(\pi_1^i) = I_{e_2}(\pi_2^i)\big) && \Rightarrow\\
&\big(\forall.E_{e_1}(\pi_1^i) = E_{e_2}(\pi_2^i) \wedge O_{e_1}(\pi_1^i) = O_{e_2}(\pi_2^i)\big)
\end{aligned}$$

With the addition of the `Clone` and `Snapshot`, the measurement of enclaves does not change for two reasons. Eq. (5.3) is satisfied because the measurements of the children are copied over from the parent and is in an equivalent enclave state as the parent. In addition, because each enclave child executes in a way that is identical to the parent without `Clone`, the child enclave e_c is still deterministic up to the inputs $I_{e_c}(\sigma)$.

Integrity The second property, integrity, states that the enclave program's execution cannot be affected by the adversary beyond the use of inputs I_e at each step and initial state $E_e(\pi_1^0)$, formalized as Eq. (5.5).

$$\begin{aligned}
&\forall \pi_1, \pi_2. && \quad (5.5)\\
&\big(E_{e}(\pi_1^0) = E_{e}(\pi_2^0) && \wedge\\
&\forall i.(curr(\pi_1^i) = e) \iff (curr(\pi_2^i) = e) && \wedge\\
&\forall i.(curr(\pi_1^i) = e) \Rightarrow I_{e}(\pi_1^i) = I_{e}(\pi_2^i)\big) && \Rightarrow\\
&\big(\forall.E_{e}(\pi_1^i) = E_{e}(\pi_2^i) \wedge O_{e}(\pi_1^i) = O_{e}(\pi_2^i)\big)
\end{aligned}$$

`Clone` creates a logical copy of the enclave whose behavior matches the parent enclave had it not been cloned and thus clone does not affect the integrity of the enclave. `Snapshot` freezes the enclave state and thus does not affect the integrity vacuously because the state of e after calling snapshot does not change until its destruction.

Confidentiality Lastly, the confidentiality property states that given the same enclave program with different secrets represented by e_1 and e_2 in traces π_1 and π_2 respectively, if the adversary starts in the initial state $A_{e_1}(\pi_1[0])$ and the protected enclave(s) e_1 (and e_2) is operated with a (potentially malicious) sequence of inputs I_{e_1}, the adversary should not learn more than what's provided by the observation function *obs* and hence its state $A_{e_1}(\sigma)$ and $A_{e_1}(\sigma)$ should be the same. The fourth line of Eq. (5.6) requires that any changes by the protected enclave e does not affect the observations made by the adversary in the next step. This is to avoid spurious counter examples where secrets leak through obvious channels such as the enclave output, which is a bug in the enclave program as explained in Section 5.4.3.

$$\begin{aligned}
&\forall \pi_1, \pi_2. \\
&\big(A_{e_1}(\pi_1^0) = A_{e_2}(\pi_2^0) && \wedge \\
&\forall i.(curr(\pi_1^i) = curr(\pi_2^i) \wedge I_{e_1}(\sigma_1^i) = I_{e_2}(\sigma_2^i)) && \wedge \\
&\forall i.(curr(\pi_1^i) = e) \Rightarrow obs(\pi_1^{i+1}) = obs(\pi_2^{i+1})\big) && \Rightarrow \\
&\big(\forall . A_{e_1}(\pi_1^i) = A_{e_2}(\pi_2^i)\big)
\end{aligned} \tag{5.6}$$

`Snapshot` alone clearly does not affect the confidentiality of the enclave. `Clone` on the other hand, also does not affect confidentiality because it creates a logical duplicate of an enclave. Had the adversary been able to break the confidentiality of the child e_c, it should have been able to break confidentiality of the parent e_p because both should behave in the same way given the same sequence of input.

5.5.2 Cerberus Platform Invariants

Proving the SRE property on TAP$_C$ requires a few key additional platform inductive invariants. Although the following list is not exhaustive, it provides a summary of the difference between the invariants in the base TAP model and the TAP$_C$ model and explains what precisely makes the other sharing models more difficult to verify. The invariants are typically over the two traces π_1 and π_2 in the properties previously mentioned. However, there are single-trace properties, and unless otherwise noted, it is assumed that single-trace properties defined over a single trace π hold for both traces π_1 and π_2 in the properties.

Memory Sharing As explained earlier, allowing the sharing of memory weakens the constraint that memory is strictly isolated. This means that the memory readable and executable by an enclave can either belong to itself or its root snapshot. This is true for the entrypoints of the enclave and the mapped virtual addresses. These are described as Eq. (5.7) and Eq. (5.8) respectively.

Eq. (5.7) states that all enclaves have an entrypoint that belongs to e itself or its snapshot $\pi^i.\mathcal{M}^{RS}[e]$.

$$\begin{aligned}
\forall e \in \mathcal{E}_{id}, \forall i.\Big(valid(e) && \Rightarrow \\
\big(\pi^i.o[\pi^i.\mathcal{M}^{EP}[e]] = e && \vee \\
\pi^i.o[\pi^i.\mathcal{M}^{EP}[e]] = \pi^i.\mathcal{M}^{RS}[e]\big)\Big)
\end{aligned} \tag{5.7}$$

Eq. (5.8) states that every enclave e whose physical address $p \in PA$ corresponding to virtual address $v \in VA$ in the page table that is mapped $\mathit{mapped}_e(\pi[i].a_{PA}[v])$[9] either belongs to e itself or the root snapshot $\pi^i.\mathcal{M}^{RS}[e]$.

To illustrate the potential complexity of the capped and arbitrary memory sharing models, the antecedent of this invariant would need to existentially quantify over all the possible snapshot enclaves that that own the memory as opposed to the current two (the enclave itself or its root snapshot). This would introduce an alternating quantifier[165] in the formula, making reasoning with SMT solvers difficult.

$$\begin{aligned}\forall e \in \mathcal{E}_{id}, v \in VA, \forall i. \Big(\big(valid(e) \wedge mapped_e(\pi^i.a_{PA}[v]) \big) \Rightarrow \\ \big(\pi^i.o[\pi^i.a_{PA}[v]] = e \quad \vee \\ \pi^i.o[\pi^i.a_{PA}[v]] = \pi^i.\mathcal{M}^{RS}[e] \big) \Big)\end{aligned} \tag{5.8}$$

Lastly, memory that is marked free for an enclave e is owned by that enclave itself, represented by Eq. (5.9).

$$\forall e \in \mathcal{E}_{id}, p \in PA, \forall i. \Big(\pi^i.\mathcal{M}^{PAF}[e][p] \Rightarrow \pi^i.o[p] = e \Big) \tag{5.9}$$

Snapshots The next invariants relate to snapshot enclaves.

First, the root snapshot of an enclave is never itself, represented by Eq. (5.10).

$$\forall e \in \mathcal{E}_{id}, \forall i. (\pi^i.\mathcal{M}^{RS}[e] \neq e) \tag{5.10}$$

Snapshots also do not have root snapshots Eq. (5.11). This invariant reflects the property that the root snapshot to ancestor enclave relationship has a height of at most 1. This is stated as all enclaves that are snapshots have a root snapshot reference pointing to the invalid enclave ID e_{inv}.

$$\begin{aligned}\forall e \in \mathcal{E}_{id}, \forall i. \Big(\big(valid(e) \wedge \pi^i.\mathcal{M}^{IS}[e] \big) \quad \Rightarrow \\ \pi^i.\mathcal{M}^{RS}[e] = e_{inv} \Big)\end{aligned} \tag{5.11}$$

Next, if an enclave has a root snapshot that is not invalid
(i.e. $\pi^i.\mathcal{M}^{RS}[e] \neq e_{inv}$), then the root snapshot is a snapshot and the child count is positive. This is represented as Eq. (5.12).

$$\begin{aligned}\forall e \in \mathcal{E}_{id}, \forall i. \Big(valid(\pi^i.\mathcal{M}^{RS}[e]) \Rightarrow \\ \big(\pi^i.\mathcal{M}^{IS}[\pi^i.\mathcal{M}^{RS}[e]] \quad \wedge \\ \pi^i.\mathcal{M}^{CC}[\pi^i.\mathcal{M}^{RS}[e]] > 0 \big) \Big)\end{aligned} \tag{5.12}$$

[9]mapped is a function that returns whether a physical address is mapped in enclave e and is equivalent to the *valid* function in the previous work [196]

The last notable invariant says that the currently executing enclave cannot be a snapshot as described in Eq. (5.13).

$$\forall e \in \mathcal{E}_{id}, \forall i.(\neg \pi[i].\mathcal{M}^{IS}[\pi[i].e_{curr}]) \tag{5.13}$$

Coming up with the exhaustive list of inductive invariants for TAP$_C$ took a majority of the verification effort.

5.6 Implementation in RISC-V Keystone

This section implements Cerberus on Keystone to show feasibility. Keystone is an open-source framework for building enclave platforms on RISC-V processors. Keystone implements the platform operations $\mathcal{O}_{base}$ in high-privileged firmware called security monitor. This work implemented additional `Snapshot` and `Clone` based on the specification. All fields of the enclave metadata live within the security monitor memory. The metadata was extended with the variables corresponding to $\mathcal{M}^{IS}$, $\mathcal{M}^{CC}$, $\mathcal{M}^{RS}$, and $\mathcal{M}^{PAF}$.

The implementation complies with the assumptions of the model described in Section 5.4.2. First, Keystone enclave operations are atomic operations, which update the system state only when the operation succeeds. Second, the implementation leverages Keystone's free memory module to achieve deterministic memory allocation for copy-on-write.

For memory isolation, Keystone uses a RISC-V feature called Physical Memory Protection (PMP) [217], which allows the platform to allocate a contiguous chunk of physical memory to each enclave. When an enclave executes, the corresponding PMP region is activated by the security monitor. The weakened constraints (i.e., Eq. (5.8)) were implemented by activating the snapshot's memory region when the platform context switches into the enclave.

In the model, the platform would need to handle the copy-on-write. In Keystone, an enclave can run with supervisor privilege, which allows the enclave to manage its page table. This was very useful because the platform does not need to understand the virtual memory mapping of the enclave. Letting the enclave handle its write faults does not hurt the security because the permissions on physical addresses are still enforced by the platform. One implementation challenge was that the enclave handler itself would always trigger a write fault because the handler requires some writable stack to start execution. This work implements a stack-less page table traverse, which allows the enclave to remap the page triggering the write fault without invoking any memory writes. The final copy-on-write handler is similar to an on-demand fork [231].

5.7 Evaluation

The evaluation goals are as follows:

- **Verification Results:** Show that the incremental verification approach enables fast formal reasoning on enclave platform modification.

- **Start-up Latency:** Show that Cerberus interface can be used with process-creation system calls to reduce the start-up latency of enclaves
- **Computation Overhead:** Show that the copy-on-write implementation does not incur significant computation overhead
- **Programmability:** Show that Cerberus provides a programmable interface, which can be easily used to improve the end-to-end latency of server enclave programs.

The performance evaluation used a SiFive's FU540 [81] processor running at 1 GHz and an Azure DC1s_v3 VM instance with an Intel® Xeon® Platinum 8370C running at 2.4 GHz to run Keystone and SGX workloads respectively. Each experiment was averaged over 10 trials.

5.7.1 Verification Results

The TAP$_C$ model and proofs can be found at `https://github.com/anonymous1721/TAPC.git`.

Porting TAP from Boogie to UCLID5. One other contribution of this work includes the port of the original TAP model from Boogie [23] to UCLID5 [210, 181] (See [67]). UCLID5 is a verification toolkit designed to model transition systems modularly, which provides an advantage over the previous implementation written in the software-focused verification IR Boogie. UCLID5 is advantageous over other state-of-the-art tools [112, 54, 22, 205] because of modularity and because it provides flexibility in modeling systems both operationally and axiomatically. This effort took three person-months working approximately 25 hours a week to finish.

Verifying TAP$_C$. The modeling and verification took roughly three person-months to write the extensions to the TAP model and verify using a scalable approach. This time is substantially less than it would have taken to rebuild the model from scratch without an existing abstraction.

Figure 5.6 shows the number of procedures #pn, number of (uninterpreted) functions #fn, number of annotations #an (which include pre- and post-conditions, loop invariants, and system invariants), the number of lines of code #ln. The last column shows the verification time which includes the time it took UCLID5 to generate verification conditions and print them out in SMTLIB2.0 and verify them using Z3/CVC4[10]. The time discrepancy between the original proofs [196] and the ones in this effort can be explained by the way all the verification conditions are generated as SMTLIB on disk before verifying as a way to use other SMT solvers. Also, this work uses UCLID5 instead of Boogie [23]. The number of lines for `Snapshot` and `Clone` is 1110, which means only 489 lines were used to extend the existing TAP operations and platform model.

Despite the added complexity, each operation for each proof took only a few minutes to verify individually as shown in the last column of Figure 5.6. This demonstrates that the incremental verification methodology is practical and consequently reduces the overall time to verify additional operations at a high level.

[10]For one of the properties, Z3 would get stuck but CVC4 didn't.

Model/Proof	**Size** #pr	#fn	#an	#ln	**Verif. Time (s)**
TAP Models					
`TAP`	43	14	225	2100	140
`Integrity`	2	0	52	525	285
`Mem. Conf`	3	0	44	838	342
TAP$_C$ Models					
`TAP`	45	16	466	3689	1380
`Integrity`	2	0	109	937	934
`Mem. Conf`	3	0	119	1307	944

Figure 5.6: Model Statistics and Verification Times

The results show that the single-sharing model makes formal encoding and verification practical. Also, the results confirm that introducing invariants with alternating existential quantifiers in the models degraded the verification time and would likely do the same for alternative models. These attempts heavily influenced the design decisions.

5.7.2 Start-up Latency

This section implements `fork` and `clone` system calls based on Cerberus to show the efficacy of the Cerberus interface. When an enclave program invokes the system calls, it calls `Snapshot` to create an immutable image and cooperates with the OS to clone two enclaves from the snapshot using `Clone`. This section compares the `fork` latency on two platforms: SGX-based Graphene [209] (now Gramine Linux Foundation project [74]) and RISC-V Keystone [110] with Cerberus. Figure 5.7 shows the program that calls `fork` after allocating memory with `SIZE`.

The baseline (Graphene-SGX) latency increases significantly as the allocation size increases (Figure 5.8). With a 400 MB buffer, it takes more than 6 seconds to complete. Also, each enclave will take 400 MB of memory at all times, even when most of the content is identical until one of the enclaves writes. With Cerberus, the latency does not increase with respect to the allocation size, because the implementation is not copying any of the parent's memory, including the page tables. It only took 23 milliseconds to fork on average, with a standard deviation of 16 microseconds.

5.7.3 Computation Overhead

This section measures the computation overhead incurred by CoW invocation. The experiment uses the RV8 [169] benchmark to see the overhead for various memory sizes and access patterns RV8 consists of 8 simple applications that perform single-threaded computation. The experiment omitted `bigint` because of a known bug in Keystone [11]. Since RV8 does not use `fork`, the exper-

[11]The provided toolchain was unable to compile C++ binary correctly

```c
int main() {
  char* buf = malloc(SIZE);
  clock_t start = clock();
  if (!fork()) {clock_t end = clock();} // child
  else { return; }                      // parent
}
```

Figure 5.7: C code to measure `fork` latency

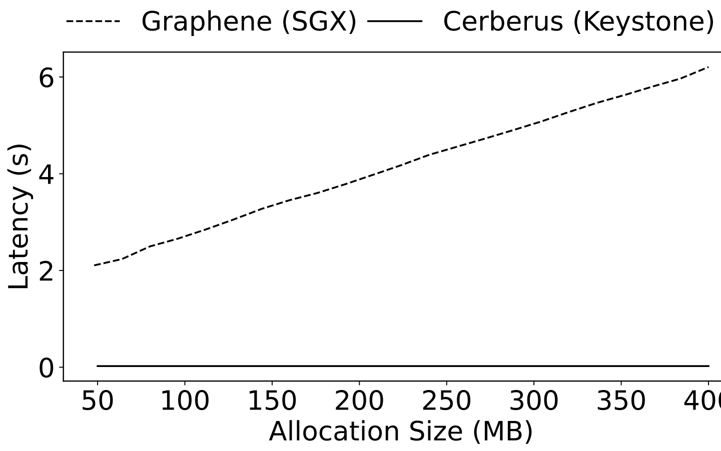

Figure 5.8: The latency of `fork` with respect to the size of the allocated memory.

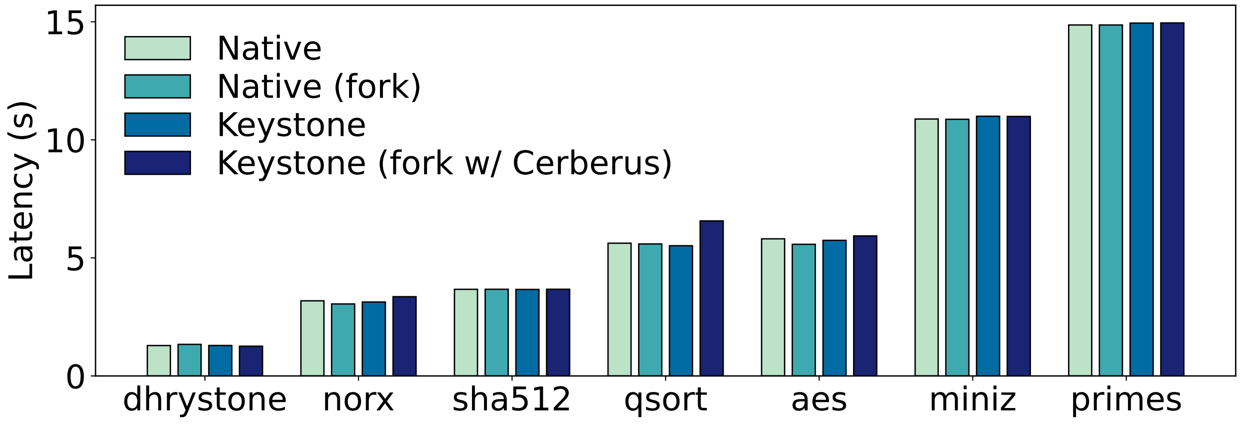

Figure 5.9: Computation Overhead on RV8. Native: native execution of the original RV8, Native (fork): native execution of the modified RV8 with `fork`, Keystone: enclave execution of the original RV8, and Keystone (fork w/ Cerberus): enclave execution of the modified RV8 with Cerberus.

iment modified RV8 such that each of the benchmark forks before the computation begins. Note that all programs in the benchmark allocate a large buffer, initialize it, and start the computation. Thus, the experiment inserted a fork right after the allocation so that the computation triggers CoW on various addresses depending on the memory access pattern.

As shown in Figure 5.9, the average computation overhead of copy-on-write memory over Keystone was only 3.9%. The worst overhead was 19.0% incurred in `qsort`, which uses the largest memory (about 190 MiBs). Memory sharing generally does not benefit such workloads with a massive dissimilar buffer.

5.7.4 Programmability

This section demonstrates a usage of the Cerberus interface in enclave programming by showing how server programs can leverage memory sharing to improve their end-to-end performance.

Although `Snapshot` or `Clone` are not directly related to `fork` or `clone`, their behavior maps well with `Snapshot` and `Clone`. For example, those system calls create a new process with the same virtual memory, which can be mapped to `Clone` and optimized by `Snapshot`. Thus, two students were provided with the modified `fork` and `clone` that use Cerberus interface and asked to modify the server programs to leverage memory sharing.

A student modified darkhttpd, a single-threaded web server, to fork processes to handle new HTTP requests inside the event loop. The modification allowed darkhttpd to serve multiple requests concurrently and continue listening for new requests. The experiment measures the latency of an HTTP request using `wget` to fetch 0.5 MB of data. The resulting program incurs only a 2.1x slowdown over the native (non-enclave) execution, in contrast to a 33x slowdown in corresponding Intel SGX implementation (the same program ran with Graphene). The 2.1x overhead is mainly due to the slow I/O system calls, which are a well-known limitation of enclaves [218, 110].

Another student implements a simple read-only database server application using *Sqlite3*, a single-file SQL library that supports in-memory and file-based databases. The resulting program serves each query with a fresh child created by `fork`. The experiment measures the latency of 1,000 `SELECT` queries served by separate enclaves. The resulting program incurs a 36x slowdown over the native execution, compared to a 262x slowdown in corresponding Intel SGX implementation. In SGX, Darkhttpd experiences more overhead than Sqlite3 because Sqlite3 has more data to copy over (i.e., the entire in-memory database). The 36x slowdown is mainly due to limited concurrency in Keystone: since Keystone implements memory isolation with a limited number of PMP entries, it can support only up to 3-4 concurrent enclaves. The limitation is not an inherent limitation of Cerberus.

Both students did not have any difficulties allowing enclaves to share memory, because they were already familiar with the expected behavior of the system calls. However, they did not know the codebase of Darkhttpd nor Sqlite3 prior to work. Darkhttpd required modification of less than 30 out of 2,900 lines of code, which took less than ten person-hours, and Sqlite3 consists of 103 lines of code, which took less than twenty person-hours. The experiment shows that programmers can use the Cerberus extension easily to improve the end-to-end performance of server programs.

5.8 Implications and Limitations

Verifying the Implementation This work does not verify the implementation of Cerberus in Keystone. Unsurprisingly, any discrepancy between the model and the implementation can make the implementation vulnerable. In particular, the enclave page table is abstracted as enclave metadata in TAP and TAP$_C$, where it is a part of memory Π in practice. Cerberus in Keystone does not create any security holes because the page table management is trusted (the enclave manages it). However, this does not imply that the same argument applies to the other implementations. To formally verify the implementation, one can construct the model for Keystone implementation and do the refinement proof to show that the model refines the TAP model as described by Subramanyan *et al.* [196].

In-Enclave Isolation Instead of modifying the platform, a few approaches [184, 4, 127, 105] use *in-enclave* isolation mechanisms to create multiple security domains within a single enclave. However, security guarantees of such solutions rely on the formal properties of not only the enclave platform but also the additional techniques used for the isolation. For example, the security of software fault isolation (SFI) [214] based approaches [184, 4] depends on the correctness and robustness of the SFI techniques including the shared software implementation and the compiler, which should be formally reasoned together with the enclave platform. Thus, such approaches will result in a significant amount of verification efforts.

5.9 Summary

This chapter showed how to formally reason about modifying the enclave platform to allow memory sharing. The chapter introduces the single-sharing model, which can support secure and efficient memory sharing of enclaves. The chapter also proposed two additional platform operations similar to existing process-creation system calls. To formally reason about the security properties of the modification, a generic formal specification was defined by incrementally extending an existing formal model. The incremental verification allowed quick proof of the security guarantees of the enclave platform. The implementation of the extension on Keystone open-source enclave platform brought significant performance improvement to server enclaves.

Chapter 6

Conclusion and Future Work

As cloud computing gains more popularity, security and privacy have been the number one concern for both the users and the service providers. Academia and industry are considering the trusted execution environment (TEE) as one of the most promising solutions, as it provides better security than software-based isolation and higher performance than cryptography-based privacy-preserving schemes. However, designing and building TEEs are dictated by a small number of companies, leaving only a small room for researchers to verify or improve.

This thesis addressed a few research challenges of designing and building secure TEEs. Section 6.1 summarizes the contribution of this thesis. Then, Section 6.2 depicts future research opportunities based on the lessons learned in this thesis.

6.1 Contributions

Chapter 3 motivated more research on TEE threat models by showing how an off-chip side-channel attack can break the confidentiality of Intel SGX, an existing commercial TEE. While this thesis admitted that such side channels are hard to eradicate, it discussed a fundamental question of why side-channel attacks even matter. As Intel SGX does not defend against side channels by design, the attack leverages the page table management capability of the attacker that SGX overlooks. Thus, Chapter 3 urged TEEs to consider side-channel attacks, not to mention other ones [220, 35]. Most importantly, the chapter motivated the need for a better TEE design that can adapt to various threat models.

Chapter 4 presented the Keystone framework for building TEEs. The chapter explained why existing vendor TEEs are inefficient for various optimizations. Then, the chapter introduced a *customizable* TEE framework instead of a TEE design with a fixed threat model. Based on the common characteristics that most of the TEEs share, Keystone allows modular software extensions to add extra functions and security guarantees to the TEE. The modular design makes design space exploration and trade-off analysis effective while keeping the trusted computing base small. Keystone allows researchers to customize their TEEs based on various threat models, workloads, and performance requirements.

Chapter 5 suggested a modification of TEE to support secure memory sharing of enclaves. The chapter showed why a TEE needs memory sharing and why existing solutions fail to provide a formally verified design. Thus, the chapter suggested *incremental verification*, which can quickly verify the security properties of the TEE modification based on an existing abstract model. Allowing the single-sharing model with simple additional operations enables efficient memory sharing while keeping the entire design formally reasonable. The verified design was implemented in Keystone introduced in Chapter 4, which improved the performance of server applications.

To summarize, this thesis showed why we need to build a TEE with the right threat model, how to build a TEE framework such that one can customize their TEE based on their needs, and how to improve TEE performance while keeping the security properties. The following section discusses future research directions based on the progress of this thesis.

6.2 Future Work

Although TEEs provide a smaller attack surface via hardware-based isolation, they still suffer from a few security issues.

- **Side-Channel Attacks and Mitigations.** Despite the efforts to thwart them, side-channel attacks have been a continuous threat to TEEs. As Chapter 3 has shown, side channels can exist everywhere, regardless of whether it's hardware or software, open-source or closed-source, and even formally verified or not. TEEs cannot thwart all existing side-channel attacks, yet they can reduce the risk such that it has a higher bar and lower bandwidth of leakage even with a successful attack. Promising directions include fundamental defenses via hardware-software co-design [56, 141, 50], formal reasoning on side-channel attacks [40, 196], and data-oblivious computation techniques [227, 225].

- **Trusted Hardware.** Most security properties of TEEs rely on *trusted hardware* designed and manufactured by trusted parties. In practice, this makes a TEE rely on design decisions and implementation of a company. There are already substantial efforts to fully open-source hardware, including the silicon root of trust [147], processors [16, 38], and IPs [166]. Open-sourcing hardware will allow the community to inspect and verify the implementation, leading to more trustworthy hardware. However, fully trusting hardware will require more research on the entire trusted chip-making process, including synthesis, physical design, fabrication, and packaging.

- **Formal Verification.** As Chapter 5 depicted, formal methods are a powerful tool to verify the functional correctness of the system and high-level security properties like confidentiality and integrity. However, it usually involves laborious manual efforts such as writing formal models, reasoning about invariants, and applying various proof techniques. The verification efforts can fade away with more automated verification techniques such as automated synthesis [229, 63], push-button verification [137, 136], and solver-aided languages [205]. Another issue is that it is hard to reason about the TEE as a whole. A TEE involves not only software but hardware components with various microarchitectural details. As a promising approach, integration verification [62] models the hardware-software interface to verify an end-to-end system, including hardware and software.

Currently, Keystone is widely used in academia as a framework for prototyping various ideas in TEEs. However, there are still many research challenges and potential design improvements.

- **Scalable Memory Isolation.** Keystone relies on Physical Memory Protection (PMP) in standard RISC-V ISA to implement hardware-based isolation. Because each enclave requires at least one PMP entry, the number of PMP entries in the platform limits the number of enclaves in a system. Penglai Enclave [64] suggested using Guarded Page Table to enable page-granularity isolation, which relies on virtual memory with minimal hardware extension. Such an approaches will make memory isolation scalable, but at the cost of the size and complexity of the platform and trusted firmware (i.e., security monitor). There are efforts to address the issue by incorporating software-based isolation with hypervisor extension leveraging two-stage address translation, which will require additional hardware mechanisms and software verification to strengthen the TCB.
- **Efficient Memory Encryption.** Keystone offers software-based memory encryption and integrity protection [11], which incurs significant performance degradation. The performance degradation is mainly due to frequent page swapping caused by a small on-chip memory to store plain data. One can improve the performance by (1) expanding on-chip memory and using specialized hardware accelerators to speed up the page encryption, decryption, and swapping; or (2) a dedicated memory encryption engine baked in the memory controller. The lack of an open-source, performant memory controller makes the former option more attractive.

Finally, there are research opportunities around building TEEs and TEE-based applications, which will allow more TEE adoption.

- **Distributed Computation.** As more TEEs target cloud rather than mobile or desktop environments, there will be a push to protect distributed applications with TEEs. Broad adoption of TEEs in distributed computing will require efficient communication between distributed TEEs, secure key management, efficient encryption and authentication techniques for storage and memory, and TEE migration techniques.
- **Programming Model.** As Section 2 describes, TEE programs can only leverage a narrow and limited interface. Thus, the programmers often need to completely rewrite the programs, which can entail manual efforts and human errors. TEE-aware programming languages and compilers can reduce the programming burden. Also, they will enforce security decisions to be made by the programmers, reducing the risk of vulnerabilities caused by human errors.

Bibliography

[1] *ADVANCED TRUSTED ENVIRONMENT: OMTP TR1*. `http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf`. Last accessed: January 18, 2022. 2009.

[2] *AES*. `https://github.com/B-Con/crypto-algorithms`. 2015.

[3] Shaizeen Aga and Satish Narayanasamy. "InvisiMem: Smart Memory Defenses for Memory Bus Side Channel". In: *Proc. of International Symposium on Computer Architecture (ISCA)*. 2017.

[4] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. "Chancel: efficient multi-client isolation under adversarial programs". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2021.

[5] Sidney Amani et al. "Cogent: Verifying High-Assurance File System Implementations". In: *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2016.

[6] *AMD Secure Encrypted Virtualization*. `https://developer.amd.com/amd-secure-memory-encryption-sme-amd-secure-encrypted-virtualization-sev/`. Last accessed: January 1, 2022.

[7] *AMD SEV-SNP*. `https://developer.amd.com/sev/`. 2020.

[8] S. r. Ames, R. Schell, and M. Gasser. "Security Kernel Design and Implementation: An Introduction". In: *Computer* 16.07 (1983).

[9] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. "Innovative Technology for CPU Based Attestation and Sealing". In: *Proc. of Workshop on Hardware and Architectural support for Security and Privacy (HASP)*. 2013.

[10] James P Anderson. *Computer Security Technology Planning Study*. Tech. rep. Anderson (James P) and Co Fort Washington PA, 1972.

[11] Gui Andrade, Dayeol Lee, David Kohlbrenner, Krste Asanović, and Dawn Song. "Software-Based Off-Chip Memory Protection for RISC-V Trusted Execution Environments". In: *Proc. of the Third Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2020.

[12] *Arm Confidential Compute Architecture*. `https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture`.

[13] *ARM TrustZone*. `https://www.arm.com/products/security-on-arm/trustzone`. 2013.

[14] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[15] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[16] Krste Asanović et al. "The Rocket Chip Generator". In: UCB/EECS-2016-17 (2016).

[17] N. Asokan. "Hardware-Assisted Trusted Execution Environments: Look Back, Look Ahead". In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019.

[18] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. "LibSEAL: Revealing Service Integrity Violations Using Trusted Execution". In: *Proc. of European Conference on Computer Systems (EuroSys)*. 2018.

[19] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. "ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories". In: *Proc. of International Symposium on Computer Architecture (ISCA)*. 2017.

[20] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World". In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014.

[21] Sundeep Bajikar. "Trusted platform module (tpm) based security on notebook pcs-white paper". In: *Mobile Platforms Group Intel Corporation* 1 (2002), p. 20.

[22] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2022.

[23] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLIne, Bart Jacobs, and Rustan Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Proc. of Formal Methods for Components and Objects (FMCO)*. 2005.

[24] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[25] Rachel Bennett. *How going digital has impacted cloud adoption*. `https://blogs.oracle.com/cloud-infrastructure/post/how-going-digital-has-impacted-cloud-adoption`. 2021.

[26] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. “Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019.

[27] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har’El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. “The Turtles Project: Design and Implementation of Nested Virtualization”. In: *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[28] Boolos, George S. and Burgess, John P. and Jeffrey, Richard C. “The Undecidability of First-Order Logic”. In: *Computability and Logic*. 5th ed. Cambridge University Press, 2007.

[29] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. “MI6: Secure Enclaves in a Speculative Out-of-Order Processor”. In: *Proc. of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2019.

[30] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. “Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory”. In: *Proc. of Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2017.

[31] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. “Sanctuary: ARMing TrustZone with User-space Enclaves”. In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2019.

[32] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*. 2017.

[33] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. “SecureKeeper: Confidential ZooKeeper Using Intel SGX”. In: *Proc. of International Middleware Conference (Middleware)*. 2016.

[34] Ernie Brickell, Jan Camenisch, and Liqun Chen. “Direct Anonymous Attestation”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2004.

[35] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *Proc. of USENIX Security Symposium*. 2018.

[36] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. “Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. In: *Proc. of USENIX Security Symposium*. 2017.

[37] Claudio Canella et al. “Fallout: Leaking Data on Meltdown-resistant CPUs”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019.

[38] Christopher Celio, David A. Patterson, and Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Tech. rep. UCB/EECS-2015-167. 2015.

[39] David Champagne and Ruby B. Lee. "Scalable architectural support for trusted software". In: *Proc. of International Symposium on High-Performance Computer Architecture (HPCA)*. 2010.

[40] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. "A formal approach to secure speculation". In: *Proc. of IEEE Computer Security Foundations Symposium (CSF)*. 2019.

[41] Stephen Checkoway and Hovav Shacham. "Iago attacks: why the system call API is a bad untrusted RPC interface". In: *Proc. of Architectural support for programming languages and operating systems (ASPLOS)*. 2013.

[42] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races". In: *S&P*. 2018.

[43] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. "Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu". In: *Proc. of ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. 2017.

[44] Xi Chen, Robert P Dick, and Alok Choudhary. "Operating system controlled processor-memory bus encryption". In: *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2008.

[45] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. "Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems". In: *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2008.

[46] Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. "The Cogent Case for Property-Based Testing". In: *Proc. of Workshop on Programming Languages and Operating Systems (PLOS)*. 2017.

[47] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts". In: *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019.

[48] *cloc - count lines of code*. `https://github.com/AlDanial/cloc`. 2020.

[49] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. 2016.

[50] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation". In: *Proc. of USENIX Security Symposium*. 2016.

[51] John Criswell, Nathan Dautenhahn, and Vikram Adve. "Virtual Ghost: Protecting Applications from Hostile Operating Systems". In: *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2014.

[52] *ARM Security IP CryptoIsland Family*. `https://www.arm.com/products/silicon-ip-security/cryptoisland`. Last accessed: December 2, 2019.

[53] Mark Horowitz David Lie Chandramohan A. Thekkath. "Implementing an Untrusted Operating System on Trusted Hardware". In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 2003.

[54] Leonardo De Moura and Nikolaj Bjorner. "Z3: An Efficient SMT Solver". In: *Proc. of the Theory and Practice of Software, in 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*. 2008.

[55] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[56] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments". In: *Proc. of USENIX Security Symposium*. 2020.

[57] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. "M2R: Enabling Stronger Privacy in MapReduce Computation". In: *Proc. of USENIX Security Symposium*. 2015.

[58] J.G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S.W. Smith. "Building the IBM 4758 secure coprocessor". In: *Computer* 34.10 (2001), pp. 57–66. DOI: `10.1109/2.955100`.

[59] *Ed25519*. `https://github.com/mit-sanctum/ed25519`. 2019.

[60] Dawson R. Engler. "The Exokernel Operating System Architecture". AAI0800457. PhD thesis. Cambridge, MA, USA, 1998.

[61] *Enron Email Dataset*. `https://www.cs.cmu.edu/~./enron/`. Last accessed: December 2, 2019.

[62] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. "Integration Verification across Software and Hardware for a Simple Embedded System". In: *Proc. of ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*. 2021.

[63] Grigory Fedyukovich and Rastislav Bodík. "Accelerating syntax-guided invariant synthesis". In: *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2018.

[64] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. "Scalable Memory Protection in the PENGLAI Enclave". In: *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2021.

[65] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. "Komodo: Using verification to disentangle secure-enclave hardware from software". In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 2017.

[66] Brad Fitzpatrick. "Distributed caching with memcached". In: *Linux journal* 2004.124 (2004), p. 5.

[67] Pranav Gaddamadugu. "Formally Verifying Trusted Execution Environments with UCLID5". MA thesis. EECS Department, University of California, Berkeley, Aug. 2021.

[68] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". In: *Journal of Cryptographic Engineering* (2018).

[69] Craig Gentry. "Computing arbitrary functions of encrypted data". In: *Communications of the ACM* 53.3 (2010), pp. 97–105.

[70] *GNU Privacy Guard*. `http://www.gnupg.org`. Last accessed: December 2, 2019.

[71] O. Goldreich, Silvio Micali, and Avi Wigderson. "How to Play ANY Mental Game". In: *Proc. of ACM symposium on Theory of computing (STOC)*. 1987.

[72] David Goltzsche et al. "EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution". In: *Proc. of Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018.

[73] Deli Gong, Muoi Tran, Shweta Shinde, Hao Jin, Vyas Sekar, Prateek Saxena, and Min Suk Kang. "Practical Verifiable In-network Filtering for DDoS defense". In: *Proc. of International Conference on Distributed Computing Systems (ICDCS)*. 2019.

[74] *Gramine*. `https://github.com/gramineproject/gramine`. 2021.

[75] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory". In: *Proc. of USENIX Security Symposium*. 2017.

[76] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Proc. of Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2016.

[77] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. "Deep Specifications and Certified Abstraction Layers". In: *Proc. of ACM Symposium on Principles of Programming Languages (POPL)*. 2015.

[78] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[79] Shay Gueron. *A Memory Encryption Engine Suitable for General Purpose Processors*. Cryptology ePrint Archive, Report 2016/204. 2016.

[80] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Cal, Ariel J. Feldman, and Edward W. Felten. "Lest We Remember: Cold Boot Attacks on Encryption Keys". In: *USENIX Security Symposium*. 2008.

[81] *HiFive Unleashed*. `https://www.sifive.com/boards/hifive-unleashed`. 2020.

[82] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. "InkTag: Secure Applications on an Untrusted Operating System". In: *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2013.

[83] R. Housley, W. Polk, W. Ford, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List Profile*. United States, 2002.

[84] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. "vTZ: Virtualizing ARM TrustZone". In: *Proc. of USENIX Security Symposium*. 2017.

[85] Andrew Huang. "Keeping Secrets in Hardware: The Microsoft Xbox^{TM} Case Study". In: *Proc. of International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. 2003.

[86] *Hunspell*. `http://hunspell.github.io/`. Last accessed: December 2, 2019.

[87] Galen Hunt, George Letey, and Ed Nightingale. *The Seven Properties of Highly Secure Devices*. Tech. rep. Mar. 2017. URL: `https://www.microsoft.com/en-us/research/publication/seven-properties-highly-secure-devices/`.

[88] *Hypervisor draft v0.5*. `https://github.com/riscv/riscv-isa-manual/releases/tag/draft-20191030-899457c`. 2019.

[89] *Intel Trust Domain Extensions*. `https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf`. Last accessed: January 1, 2022.

[90] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf`. Last accessed: December 2, 2019.

[91] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. "SGX-Bomb: Locking Down the Processor via Rowhammer Attack". In: *Proc. of Workshop on System Software for Trusted Execution (SysTEX)*. 2017.

[92] Mohit Kumar Jangid, Guoxing Chen, Yinqian Zhang, and Zhiqiang Lin. "Towards Formal Verification of State Continuity for Enclave Programs". In: *Proc. of USENIX Security Symposium*. 2021.

[93] *JKI Inc. JLA320A*. `https://www.jkic.co.kr/ddr4-protocol-analyzer`. Last accessed: December 2, 2019.

[94] Simon Paul Johnson. *Intel SGX and Side Channels*. `https://www.intel.cn/content/www/cn/zh/developer/articles/technical/intel-sgx-and-side-channels.html`. Last accessed: January 18, 2022.

[95] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. *Intel Software Guard Extensions: EPID Provisioning and Attestation Services*. 2016.

[96] David Kaplan. *AMD SEV-ES*. `https://developer.amd.com/sev/`. 2017.

[97] Sagar Karandikar et al. "Firesim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud". In: *Proc. of International Symposium on Computer Architecture (ISCA)*. 2018.

[98] Pierre Selwan Ken Irving. *Revolutionizing the Computing Landscape and Beyond*. `https://content.riscv.org/wp-content/uploads/2018/12/RISC-V-MultiCore-Secure-Boot-Ken-Irvining-and-Pierre-Selwan.pdf`. Dec. 2018.

[99] Steven L Kinney. *Trusted platform module basics: using TPM in embedded systems*. Elsevier, 2006.

[100] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors". In: *Proc. of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018.

[101] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 2009.

[102] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2019.

[103] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. "TrustLite: A Security Architecture for Tiny Embedded Devices". In: *Proc. of European Conference on Computer Systems (EuroSys)*. 2014.

[104] Oliver Kömmerling and Markus G Kuhn. "Design Principles for Tamper-Resistant Smartcard Processors." In: *Proc. of Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology (WOST)*. 1999.

[105] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "SGXBOUNDS: Memory safety for shielded execution". In: *Proc. of the Twelfth European Conference on Computer Systems (EuroSys)*. 2017.

[106] Andrew Law, Chester Leung, Rishabh Poddar, Raluca Ada Popa, Chenyu Shi, Octavian Sima, Chaofan Yu, Xingmeng Zhang, and Wenting Zheng. "Secure collaborative training and inference for xgboost". In: *Proc. of the Workshop on Privacy-Preserving Machine Learning in Practice (PPMLP)*. 2020.

[107] Ilia Lebedev, Kyle Hogan, and Srinivas Devadas. “Secure Boot and Remote Attestation in the Sanctum Processor”. In: *Proc. of IEEE Computer Security Foundations Symposium (CSF)*. 2018.

[108] Ilia Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanović, Dawn Song, and Srinivas Devadas. “Sanctorum: A lightweight security monitor for secure enclaves”. In: *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019.

[109] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. “An Off-Chip Attack on Hardware Enclaves via the Memory Bus”. In: *Proc. of USENIX Security Symposium*. 2020.

[110] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *Proc. of European Conference on Computer Systems (EuroSys)*. 2020.

[111] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing”. In: *Proc. of USENIX Security Symposium*. 2017.

[112] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Proc. of Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 2010.

[113] Mingyu Li, Yubin Xia, and Haibo Chen. “Confidential Serverless Made Efficient with Plug-in Enclaves”. In: *Proc. of International Symposium on Computer Architecture (ISCA)*. 2021.

[114] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena. “DroidVault: A Trusted Data Vault for Android Devices”. In: *Proc. of International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2014.

[115] J. Liedtke. “On Micro-kernel Construction”. In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 1995.

[116] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. “Teechain: A Secure Payment Network with Asynchronous Blockchain Access”. In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 2019.

[117] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *Proc. of USENIX Security Symposium*. 2018.

[118] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks Are Practical”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2015.

[119] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. "PHANTOM: Practical Oblivious Computation in a Secure Processor". In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2013.

[120] Saara Matala, Thomas Nyman, and N. Asokan. *Historical insight into the development of Mobile TEEs*. `http://blog.ssg.aalto.fi/2019/06/historical-insight-into-development-of.html`. 2019.

[121] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. "ROTE: Rollback Protection for Trusted Execution". In: *Proc. of USENIX Security Symposium*. 2017.

[122] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. "TrustVisor: Efficient TCB Reduction and Attestation". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2010.

[123] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. "Flicker: An execution infrastructure for TCB minimization". In: *Proc. of European Conference on Computer Systems (EuroSys)*. 2008.

[124] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. "Spectre is here to stay: An analysis of side-channels and speculative execution". In: *arXiv preprint arXiv:1902.05178* (2019).

[125] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. "Intel Software Guard Extensions Support for Dynamic Memory Management Inside an Enclave". In: *Proc. of Workshop on Hardware and Architectural support for Security and Privacy (HASP)*. 2016.

[126] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative Instructions and Software Model for Isolated Execution". In: *Proc. of Workshop on Hardware and Architectural support for Security and Privacy (HASP)*. 2013.

[127] Marcela S Melara, Michael J Freedman, and Mic Bowman. "EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments". In: *arXiv preprint arXiv:1907.13245* (2019).

[128] James Langston. *Enhancing the Scalability of Memcached*. `https://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached`. Last accessed: December 2, 2019.

[129] Ralph C Merkle. "A digital signature based on a conventional encryption function". In: *Proc. of Conference on the Theory and Applications of Cryptographic Techniques (CRYPTO)*. 1987.

[130] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. "Proof of Luck: an Efficient Blockchain Consensus Protocol". In: *Proc. of Workshop on System Software for Trusted Execution (SysTEX)*. 2016.

[131] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. "Oblix: An efficient oblivious search index". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2018.

[132] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *Proc. of Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2017.

[133] Keaton Mowery, Michael Wei, David Kohlbrenner, Hovav Shacham, and Steven Swanson. "Welcome to the Entropics: Boot-time entropy in embedded devices". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2013.

[134] *MultiZone Hex Five Security*. `https://hex-five.com/`. 2020.

[135] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2020.

[136] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. "Serval: Scaling Symbolic Evaluation for Automated Verification of Systems Code". In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 2019.

[137] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. "Hyperkernel: Push-Button Verification of an OS Kernel". In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 2017.

[138] Khang T Nguyen. *Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family*. Feb. 2016.

[139] Rajesh Nishtala et al. "Scaling Memcache at Facebook". In: *Proc. of the Symposium on Networked Systems Design and Implementation (NSDI)*. 2013.

[140] *NLTK Data 3.4.5 Documentation*. `https://www.nltk.org/data.html`. Last accessed: December 2, 2019.

[141] Hyunyoung Oh, Adil Ahmad, Seonghyun Park, Byoungyoung Lee, and Yunheung Paek. "TRUSTORE: Side-Channel Resistant Storage for SGX Using Intel Hybrid CPU-FPGA". In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020.

[142] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. "Oblivious Multi-Party Machine Learning on Trusted Processors". In: *Proc. of USENIX Security Symposium*. 2016.

[143] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *Proc. of USENIX Anual Technical Conference (ATC)*. 2018.

[144] *Open Enclave SDK*. `https://openenclave.io/sdk/`. 2020.

[145] *Open mobile terminal platform - Wikipedia*. `https://en.wikipedia.org/wiki/Open_Mobile_Terminal_Platform`. Last accessed: January 18, 2022.

[146] *Open Portable TEE*. `https://www.op-tee.org/`. 2020.

[147] *OpenTitan: Open Source Silicon Root of Trust*. `https://opentitan.org/`. Last accessed: Feb 17, 2022.

[148] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. "Eleos: ExitLess OS Services for SGX Enclaves". In: *Proc. of European Conference on Computer Systems (EuroSys)*. 2017.

[149] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. "CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves". In: *Proc. of USENIX Anual Technical Conference (ATC)*. 2019.

[150] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Proc. of Cryptographers' Track at the RSA Conference (CT-RSA)*. 2006.

[151] Nate Graff Palmer Dabbelt. *SiFive's Trusted Execution Reference Platform*. `https://content.riscv.org/wp-content/uploads/2018/12/SiFives-Trusted-Execution-Reference-Platform-Palmer-Dabbelt-1-1.pdf`. Dec. 2018.

[152] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. "Memoir: Practical State Continuity for Protected Modules". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2011.

[153] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. "Bootstrapping Trust in Commodity Computers". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2010.

[154] J Thomas Pawlowski. "Hybrid Memory Cube (HMC)". In: *2011 IEEE Hot Chips 23 Symposium (HCS)*. 2011.

[155] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting Dram Addressing for Cross-CPU Attacks". In: *Proc. of USENIX Security Symposium*. 2016.

[156] Raluca Ada Popa. "MC2: A Secure Collaborative Computation Platform". In: *Proc. of the Workshop on Privacy-Preserving Machine Learning in Practice (PPMLP)*. 2020.

[157] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. "Rethinking the Library OS from the Top Down". In: *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2011.

[158] Nelly Porter and Jason Garms. *Advancing confidential computing with Asylo and the Confidential Computing Challenge*. Feb. 2019.

[159] Dan R. K. Ports and Tal Garfinkel. "Towards Application Security on Untrusted Operating Systems". In: *Proc. of Conference on Hot Topics in Security (HOTSEC)*. 2008.

[160] Christian Priebe, Kapil Vaswani, and Manuel Costa. "EnclaveDB - A Secure Database using SGX". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2018.

[161] *Proof of Elapsed Time (PoET) 1.0 Specification - Sawtooth v1.0.5 documentation*. `https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html`.

[162] *Kibra 480 Analyzer*. `http://cdn.teledynelecroy.com/files/pdf/lecroy_kibra480_datasheet.pdf`. Last accessed: December 2, 2019.

[163] *Nexus Technology MA4100*. `https://www.nexustechnology.com/products/memory-analyzers/ma4100-series-memory-analyzer/`. Last accessed: December 2, 2019.

[164] *QEMU, Open Source Processor Emulator*. `www.qemu.org`.

[165] C. R. Reddy and D. W. Loveland. "Presburger Arithmetic with Bounded Quantifier Alternation". In: *Proc. of Annual ACM Symposium on Theory of Computing (STOC)*. 1978.

[166] *RIOSLab*. `https://rioslab.org/`. Last accessed: Feb 17, 2022.

[167] *RISC-V Proxy Kernel*. `https://github.com/riscv/riscv-pk`. 2020.

[168] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly". In: *Proc. of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2007.

[169] *RV8 Benchmark*. `https://github.com/michaeljclark/rv8-bench`. 2017.

[170] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption". In: *Proc. of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2021.

[171] Samuel Weiser and Mario Werner and Ferdinand Brasser and Maja Malenko and Stefan Mangard and Ahmad-Reza Sadeghi. "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2019.

[172] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications". In: *Proc. of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2014.

[173] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. "Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification". In: *IEEE Access* 9 (2021), pp. 83067–83079.

[174] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. "ZeroTrace : Oblivious Memory Primitives from Intel SGX". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2017.

[175] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. *SGAxe: How SGX Fails in Practice*. `https://sgaxeattack.com/`. 2020.

[176] Stephan van Schaik, Alyssa Milburn, Sebastian Asterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-flight Data Load". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2019.

[177] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. "CacheOut: Leaking Data on Intel CPUs via Cache Evictions". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2021.

[178] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. "VC3: Trustworthy Data Analytics in the Cloud". In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2015.

[179] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019.

[180] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *Proc. of Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2017.

[181] Sanjit A. Seshia and Pramod Subramanyan. "UCLID5: Integrating Modeling, Verification, Synthesis and Learning". In: *Proc. of ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2018.

[182] Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. "Translation validation for a verified OS kernel". In: *Proc. of ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*. 2013.

[183] *Intel Software Guard Extensions Programming Reference*. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`. Last accessed: December 2, 2019.

[184] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX". In: *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020.

[185] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2017.

[186] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. "Panoply: Low-TCB Linux Applications With SGX Enclaves". In: *Proc. of Network and Distributed System Security Symposium (NDSS)*. 2017.

[187] Shweta Shinde, Shengi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. "BesFS: A POSIX Filesystem for Enclaves with a Mechanized Safety Proof". In: *Proc. of USENIX Security Symposium*. 2020.

[188] Rohit Sinha, Manuel Costa, Akash Lal, Nuno Lopes, Sanjit Seshia, Sriram Rajamani, and Kapil Vaswani. “A Design and Verification Methodology for Secure Isolated Regions”. In: *Proc. of ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*. 2016.

[189] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. “Moat: Verifying Confidentiality of Enclave Programs”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[190] *Software Guard eXtensions (SGX)*. `https://www.kernel.org/doc/Documentation/x86/sgx.rst`. Last accessed: January 18, 2022.

[191] *Software Guard Extenstion (SGX) SDK for Linux*. `https://github.com/intel/linux-sgx`. Last accessed: December 2, 2019.

[192] *Spell Checker Oriented Word Lists*. `http://wordlist.aspell.net/`. Last accessed: December 2, 2019.

[193] *RISC-V ISA Simulator*. `https://riscv.org/software-tools/risc-v-isa-simulator/`. Last accessed: December 2, 2019.

[194] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2013.

[195] Rob Stubbs. *Intel SGX Technology and the Impact of Processor Side-Channel Attacks*. `https://fortanix.com/blog/2020/03/intel-sgx-technology-and-the-impact-of-processor-side-channel-attacks`. Last accessed: January 18, 2022.

[196] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. “A Formal Foundation for Secure Remote Execution of Enclaves”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[197] G. Edward Suh, Charles W. O’Donnell, Ishan Sachdev, and Srinivas Devadas. “Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions”. In: *SIGARCH Comput. Archit. News* (2005).

[198] *The Trusted Execution Environment: Delivering Enhanced Security at a Lower Cost to the Mobile Market*. `https://globalplatform.org/wp-content/uploads/2018/04/GlobalPlatform_TEE_Whitepaper_2015.pdf`. Last accessed: January 18, 2022. 2011.

[199] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. “Architectural Support for Copy and Tamper Resistant Software”. In: *Proc. of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2000.

[200] Alves Tiago and Felton Don. “TrustZone: integrated hardware and software security enabling trusted computing in embedded system”. In: *Government Information Quarterly* 3.4 (2004), pp. 18–24.

[201] *Tiny SHA3*. `https://github.com/mjosaarinen/tiny_sha3/`. 2016.

[202] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. “Privado: Practical and Secure DNN Inference”. In: *ArXiv* (2018). eprint: `1810.00602`.

[203] *Torch NNs*. `https://github.com/torch/nn/tree/master/lib/THNN`. 2017.

[204] *Torch Tensors*. `https://github.com/torch/TH`. 2015.

[205] Emina Torlak and Rastislav Bodik. “Growing solver-aided languages with Rosette”. In: *Proc. of ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward!)* 2013.

[206] Bill Toulas. *New Intel chips won’t play Blu-ray disks due to SGX deprecation*. `https://www.bleepingcomputer.com/news/security/new-intel-chips-wont-play-blu-ray-disks-due-to-sgx-deprecation`. Last accessed: January 21, 2022.

[207] *Trusted execution environment - Wikipedia*. `https://en.wikipedia.org/wiki/Trusted_execution_environment`. Last accessed: January 18, 2022.

[208] *Trustonic*. `https://www.trustonic.com`. Last accessed: January 18, 2022. 2014.

[209] Chia-che Tsai, Donald E. Porter, and Mona Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *Proc. of USENIX Anual Technical Conference (ATC)*. 2017.

[210] *UCLID5: formal modeling, verification, and synthesis of computational systems*. `https://github.com/uclid-org/uclid`. Last accessed: May 13, 2022.

[211] *unifdef*. `http://dotat.at/prog/unifdef/`. 2020.

[212] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. “LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2020.

[213] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.

[214] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. “Efficient software-based fault isolation”. In: *Proc. of Symposium on Operating Systems Principles (SOSP)*. 1993.

[215] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. “Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[216] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. “Oblivious Data Structures”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014.

[217] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. `https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf`. Dec. 2021.

[218] Ofir Weisse, Valeria Bertacco, and Todd Austin. “Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves”. In: *Proc. of International Symposium on Computer Architecture (ISCA)*. 2017.

[219] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. *Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution*. Tech. rep. Technical report, 2018.

[220] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2015.

[221] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. “InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy”. In: *Proc. of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018.

[222] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. “Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World”. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2019.

[223] Andrew C Yao. “Protocols for secure computations”. In: *Proc. of IEEE Symposium on Foundations of Computer Science (SFCS)*. 1982.

[224] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack”. In: *Proc. of USENIX Security Symposium*. 2014.

[225] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. “Creating Foundations for Secure Microarchitectures With Data-Oblivious ISA Extensions”. In: *IEEE Micro* 40.3 (2020), pp. 99–107. DOI: `10.1109/MM.2020.2985366`.

[226] Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. “Elasticlave: An Efficient Memory Model for Enclaves”. In: *Proc. of USENIX Security Symposium*. 2022.

[227] Samee Zahur and David Evans. “Obliv-C: A language for extensible data-oblivious computation”. In: *Cryptology ePrint Archive* (2015).

[228] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. “Town Crier: An Authenticated Data Feed for Smart Contracts”. In: *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.

[229] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. "Synthesizing Environment Invariants for Modular Hardware Verification". In: *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2020.

[230] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. "All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption." In: *Proc. of USENIX Security Symposium*. 2016.

[231] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. "On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications". In: *Proc. of the 16th European Conference on Computer Systems (EuroSys)*. 2021.

[232] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. "Opaque: An oblivious and encrypted distributed analytics platform". In: *Proc. of the Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.