# Lessons Learned in Hardware-Assisted Operating System Security

by

Wei Huang

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering
University of Toronto

Lessons Learned in Hardware-Assisted Operating System Security

Wei Huang
Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering
University of Toronto
2023

## Abstract

Hardware security extensions are engineered to enhance software security for various purposes such as memory protection, program execution environment isolation, among others. Nevertheless, due to constrained hardware resources and the intricate nature of security goals pertaining to diverse software applications, the practical utilization of these hardware features by software developers, may diverge from those original envisioned by the hardware designers.

We examine the repurposed uses of hardware features within the field of operating system security, presenting three case studies. The first involves a light-weight memory protection system designed to guard against return-oriented programming attacks, repurposing the intended use of a memory protection hardware extension. The second case presents a suite of secure OSes aimed at bolstering application security on mobile platforms, broadening the scope of the original target audience of a mobile hardware security feature. Lastly, we discuss an attack method that augments other side-channel attacks enabling them to elude detection and mitigation by manipulating processor thermal control functionality. The adapted uses of these hardware features can lead to a variety of potential consequences. While some outcomes, such as improved efficiency or innovative defense mechanisms, may be beneficial, others could inadvertently introduce security vulnerabilities.

Through an exhaustive analysis of these three case studies, we gleaned the following insights: (1) A flexible approach to the utilization of hardware security extensions for different security purposes can yield partial security, resulting in lower and more acceptable overhead. (2) To accommodate new application security requirements, the design of the operating system can be adapted to cater to a broader range of users. (3) Hardware features that initially seem irrelevant could effectively counter the assumptions made by software defence that rely on hardware security extensions.

This thesis underscores the importance of hardware-software collaboration for achieving optimal operating system security. Through the scrutiny of three specific examples of repurposed hardware feature utilization, the study illuminates both the potential benefits and risks inherent to these interactions. Consequently, it advocates for a more holistic and cooperative approach to navigate the challenges and intricacies of current secure computing systems.

# Acknowledgements

Firstly, I would like to express my deepest gratitude to my PhD supervisor, Professor David Lie, who has been an exceptional advisor throughout my PhD journey. His influence extends beyond academic advisement, provision of technical feedbacks and moral guidance. He has consistently shown unwavering support to all of his students during both of their prosperous and challenging times, which reflects his deep-seated care, honorable character and leadership. I strive to incorporate the invaluable lessons learned from David as I navigate my future, aspiring to continually evolve as an individual.

Next, I would like to extend my thanks to my PhD thesis committee: Professor Ding Yuan, Professor Michael Stumm, Professor Mark Jeffrey, and Professor Yinqian Zhang. Their insightful advice on my thesis and their continuous encouragement have been instrumental to my completion of this thesis.

Further, I would like to acknowledge all my research collaborators, including my academic mentors, postdocs in my research group, and my fellow graduate students. Each of them, although I cannot list all the names here, has made a unique contribution to my academic journey, and for that, I am profoundly grateful.

Lastly, I would like to thank the University of Toronto and NSERC for the graduate fellowship.

# Contents

**1 Introduction** **1**

1.1 Memory Protection Extension for Defending ROP Attacks . . . . . 2

1.2 Scalable Applications in Trusted Execution Environment . . . . . 3

1.3 Hardware Features for Cache-based Side-Channel Attacks and Defence . . . . . 4

1.4 Contributions . . . . . 4

1.5 Thesis Outline . . . . . 5

**2 Background** **6**

2.1 Intel Memory Protection Extension . . . . . 6

2.2 ARM Trusted Execution Environment and Mobile Secure Payment . . . . . 9

2.3 Other ARM On-Board Hardware Related to Security . . . . . 10

2.4 Intel SGX, TSX and Cache Structures . . . . . 10

2.4.1 Intel SGX . . . . . 10

2.4.2 Intel TSX . . . . . 11

2.4.3 Intel Cache Structures . . . . . 11

2.5 Power and Thermal Management of Intel CPU . . . . . 12

**3 LMP: Light-Weighted Memory Protection with Hardware Assistance** **13**

3.1 Motivation of the LMP System . . . . . 13

3.1.1 Attack and Defence of ROP . . . . . 13

3.1.2 Contributions of the LMP System . . . . . 14

3.2 Design of the LMP System . . . . . 15

3.2.1 Threat Model . . . . . 15

3.2.2 LMP Memory Protection with the assistance of MPX . . . . . 16

3.2.3 Using the Shadow Stack . . . . . 17

3.2.4 Execute a Program with LMP . . . . . 19

3.3 Implementation of the LMP System . . . . . 21

3.3.1 LMP-enabled Compiler . . . . . 21

3.3.2 LMP Runtime . . . . . 22

3.4 Evaluation of the LMP System . . . . . 23

3.4.1 Measurement of Performance Overhead . . . . . 23

3.4.2 Code Expansion Evaluation . . . . . 25

3.4.3 Memory Overhead Comparison . . . . . 25

3.5 Related Work . . . . . 26

3.6 Summary and Discussions . . . 28

**4 Repurposing ARM TrustZone for Third-Party Applications and Roll-Back Protection 29**

4.1 Motivation of Repurposing ARM TrustZone . . . 29
4.1.1 Current ARM TrustZone Use . . . 29
4.1.2 TEE Application Model and Examples . . . 30
4.1.3 Our Approach and Contributions . . . 32
4.2 Pearl-TEE: Democratizing TrustZone using Functionality-Specific Isolation . . . 34
4.2.1 Threat Model . . . 34
4.2.2 Design Requirements . . . 35
4.2.3 Pearl-TEE Design . . . 37
4.2.4 System Overview and Operation . . . 39
4.2.5 Security Analysis . . . 45
4.3 Mint-TEE: Supporting Scalable Rollback Protection for TrustZone Applications . . . 47
4.3.1 Design Objectives . . . 47
4.3.2 System Overview . . . 48
4.3.3 Virtual Counters for persistent storage . . . 49
4.3.4 TEE App State Counters . . . 53
4.4 Implementations . . . 55
4.4.1 Pearl-TEE Prototype and Integration with Payment Services . . . 55
4.4.2 Mint-TEE Prototype and System APIs . . . 57
4.4.3 TEE Applications . . . 57
4.5 Performance Evaluation . . . 62
4.5.1 Evaluation of Pearl-TEE . . . 62
4.5.2 Evaluation of Mint-TEE . . . 66
4.6 Related Work . . . 70
4.6.1 TrustZone Isolation and TEE Application Framework . . . 71
4.6.2 Roll-back Protection . . . 72
4.7 Summary and Discussions . . . 72

**5 Aion Attacks: Manipulating Software Timers in Trusted Execution Environment 74**

5.1 Motivation and Contributions of Aion Attacks . . . 74
5.2 System Model . . . 76
5.2.1 Model of Software Timers . . . 76
5.2.2 Defender Model . . . 77
5.2.3 Timer Countermeasures . . . 79
5.3 Attack Design . . . 80
5.3.1 Aion-1: CPU Thermal and Frequency Attack . . . 81
5.3.2 Aion-2: Cache Eviction Attack . . . 82
5.4 Implementation . . . 84
5.4.1 Reference Software Timer . . . 84
5.4.2 Implementing Aion-1: CPU Thermal and Frequency Attack . . . 86
5.4.3 Implementing Aion-2: Cache Eviction Attack . . . 86

5.5 Evaluation . . . 88
5.5.1 Evaluation Purpose and Experiment Setup . . . 88
5.5.2 Aion Attack Evaluation . . . 89
5.5.3 End-to-End Attack Evaluation . . . 91
5.6 Related Work . . . 92
5.6.1 General Side-Channel Attacks . . . 92
5.6.2 General Side-channel Detection . . . 95
5.6.3 General Side-channel Attack Mitigation . . . 98
5.6.4 Cache-Based Side-Channel Attacks and Defence in SGX . . . 101
5.7 Summary and Discussions . . . 102

**6 Discussion and Conclusion** **103**
6.1 Discussion . . . 103
6.2 Conclusion and Future Work . . . 104

**Bibliography** **106**

# List of Tables

4.1 Traits of each class of TEE applications . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2 List of APIs Implemented for Secure Storage . . . . . . . . . . . . . . . . . . . . . . . 58
4.3 Time cost comparisons between TEE applications and their Baseline counterparts (time is in milliseconds). (PP: PayPal, BT: BrainTree, AP: AliPay.) . . . . . . . . . 62
4.4 Performance overhead evaluation for TEE-Chat (time is in millisecond) . . . . . . . 64
4.5 Secure payment system performance breakdown . . . . . . . . . . . . . . . . . . . . . 65
4.6 Memory usage statistics in Pearl-TEE System . . . . . . . . . . . . . . . . . . . . . . 66
4.7 Performance overhead for TEE Apps using virtual counters in comparison with the same TEE Apps without virtual counters (time in seconds) . . . . . . . . . . . . . . 67
4.8 Performance overhead for overall protection (time in seconds) . . . . . . . . . . . . . 68
4.9 Lines of code (LOC) of Different System Modules in the Implementation and Comparison with Other OSs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.10 Memory usage statistics in Mint-TEE system . . . . . . . . . . . . . . . . . . . . . . 71

5.1 Memory access latencies for different levels of the hierarchy, averaged over 10k accesses and measured in CPU cycles. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.2 The Effectiveness of the Base Side-Channel Attacks on SGX Enclave . . . . . . . . . 89
5.3 The Effectiveness of the SGX Side-Channel Attack Detector . . . . . . . . . . . . . . 89
5.4 Results of software timer readings affected by the Aion attacks. (RF: Randome Function, CT: CPU Thermal, Base: Baseline, ST: Single-Threaded, MT: Multi-Threaded, CE: Cache Eviction.) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
5.5 End-to-end evaluation of Aion attacks against existing defences. . . . . . . . . . . . . 92
5.6 Summary of cache-based side-channel attack examples . . . . . . . . . . . . . . . . . 93
5.7 Summary of side-channel defence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100

# List of Figures

2.1 An example for illustrating how MPX works to protect pointer bounds. . . . . . . . 8

3.1 The illustration of LMP shadow stacks . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 Assembly code example for instrumented function entry/exit. . . . . . . . . . . . . 19
3.3 A flow chart of how LMP system works. . . . . . . . . . . . . . . . . . . . . . . . . 20
3.4 An example of LMP instrumentation for store instruction. . . . . . . . . . . . . . . 22
3.5 LMP overhead by comparison of execution time between baseline and LMP. . . . . 24
3.6 Overhead components of LMP. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.7 Code Expansion of LMP. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.1 System overview of Pearl-TEE with multiple TEE-enabled applications. . . . . . . . 35
4.2 Mobile payment framework system design overview. . . . . . . . . . . . . . . . . . . 40
4.3 Interactions in the phase of Device Payment Flow . . . . . . . . . . . . . . . . . . . 45
4.4 Illustration of the Mint-TEE hash tree used for storing virtual counters, as it appears in shared memory. It relies on a single hardware counter $MC_p$. . . . . . . . . . . . 50
4.5 Steps taken by Mint-TEE when two file writes by different TEE Apps are performed. 52
4.6 System Overview of a TEE application framework in Mint-TEE. . . . . . . . . . . . 60
4.7 The steps used for performance comparison between the Baseline payment application and the TEE-enabled payment application . . . . . . . . . . . . . . . . . . . . . . . . 63
4.8 Batch scheduling settings and force-termination trigger rate . . . . . . . . . . . . . . 66
4.9 Evaluation of Hardware Monotonic Counter Wear-Out . . . . . . . . . . . . . . . . . 68

5.1 A model of how a software timer works and is used by an application thread. . . . . 79
5.2 Illustration of Intel cache slice and cache set structure. . . . . . . . . . . . . . . . . 80
5.3 Illustration of the Aion-2: Cache eviction attack. . . . . . . . . . . . . . . . . . . . 84
5.4 Abstracted attacking procedures to categorize different types of techniques used by cache-based side-channel attacks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

# List of Algorithms

1 Aion-1: CPU Thermal and Frequency Attack . . . . . 87
2 Aion-2: Cache Eviction Attack . . . . . 88

# Chapter 1

# Introduction

Operating system (OS) security constitutes a pivotal facet of modern computing systems, hinging on the flawless integration of hardware and software components. Numerous hardware extensions, developed over time, cater to a spectrum of purposes including parallel execution, virtualization and safeguarding sensitive data. The complex hardware security extensions are designed by hardware manufacturers to meet specific security requirements. However, in the pursuit of achieving a balance between optimal performance and robust security, software developers may repurpose these features, employing them in manners unforeseen by the original hardware designers.

The repurposed utilization of hardware features is not uncommon in software systems, and such uses may often lead to security risks. One of the representative illustrations of this is branch prediction, a feature that empowers the processor to anticipate the outcome of a conditional branch and execute the correct instruction before the actual branch is taken. This feature, designed and implemented by CPU manufacturers to enhance processing efficiency, can unexpectedly be leveraged to leak crucial information like in Spectre attack [96] – a risk unforeseen by hardware designers. Both offensive and defensive strategies frequently diverge from the original intended use of hardware extensions, culminating in outcomes that can be advantageous or deleterious. These unconventional applications of hardware features necessitate a comprehensive understanding of the implications of employing hardware in non-traditional manners and the potential repercussions on system security.

In this thesis, we introduce three sets of mechanisms as exemplars, illuminating the repurposed use of hardware features in the realm of operating system security. We delve into memory protection hardware features such as Intel's Memory Protection Extension (MPX), designed to prevent program's out-of-bound accesses, and Intel's Secure Guard Extension (SGX), which provides a secure execution environment for user-level software applications. We assess how these hardware features have been repurposed for operating system security. We further discuss hardware extensions for Trusted Execution Environments (TEEs) on mobile platforms, elaborating on their role in augmenting the security of sensitive operations under the device manufacturers' purview and how they could be extended more freely to third-party mobile applications, or more scalable virtual hardware security features. Finally, we explore the adaptations of processor settings for security analysis purposes, uncovering the feasibility of aiding side-channel attacks by adjusting these settings in manners that deviate from their initial design.

Through the proposed mechanisms, we identify disparities between the security-focused hardware extensions designed by manufacturers and the actual requirements and applications of these features by software developers. Given the relative complexity of security hardware extensions — considering their newly introduced instructions and methods of applying them — these are usually narrowed down to one or a few fundamental use cases. Consequently, software developers may need to devise novel approaches to fit their specific requirements by changing the default methods of system software using these hardware extensions. Conversely, in certain scenarios, these changes may result in undesired consequences, necessitating alterations in software defense strategies or hardware designs. In each of the three examples within this thesis, we strive to contribute to a more profound understanding of the dynamic interplay between hardware and software in the context of operating system security. By elucidating the manners in which hardware features are repurposed, we aspire to guide future research and development efforts, as well as inform policy and best practice recommendations. Ultimately, our aim is to foster a more vigilant and informed approach to leveraging hardware features in software development, thereby promoting a secure, resilient, and open computing environment for all.

We have learned lessons during our pursuit of the above goals: (a) fully protecting an operating system for security can be a complex and expensive task, even with hardware assistance, however, with a slight deviation from the original use case, partial security can be achieved with reasonable cost; (b) the target audience of a hardware security extension might be initially small, but as the range of users expands, operating system support for such adaptations might be necessary to meet new security requirements; and (c) operating system security needs for careful consideration of hardware features that may appear irrelevant but can actually negate the assumptions made by software defence with hardware security extensions.

We will briefly introduce the three cases we studied in this thesis, offering insightful revelations into the challenges and opportunities pertinent to the employment of hardware security extensions for both software defence and attacks. Collectively, these cases underline the intricacies and hurdles associated with deploying hardware security extensions for operating system security. On the whole, these cases not only accentuate the intricacies and challenges inherent in using hardware security extensions for bolstering software defense, but they also illuminate the potential advantages that careful planning and collaboration between hardware and software designers can bring.

## 1.1 Memory Protection Extension for Defending ROP Attacks

Memory corruption attacks have plagued computer systems for decades, and despite the long history and numerous proposed defenses, these attacks continue to pose a significant threat. One of the most popular attacks is the return-oriented programming (ROP) attack, which manipulates the victim thread's call stack to execute existing code fragments, called "gadgets," by chaining their return addresses. By overwriting the return addresses on the stack, the attacker can control the execution flow and perform malicious actions without injecting new code, bypassing traditional security defence. A secure and low-overhead defense against the ROP attacks remains an elusive goal for the security community. Existing solutions often involve trade-offs between inadequate protection of critical data, relying on information hiding, or resorting to incomplete and coarse-grain checking that can be bypassed by more sophisticated and well-orchestrated attacks.

In this work, we introduce a novel approach called Light-Weight Memory Protection (LMP) that leverages Intel's MPX hardware extension, originally designed for bounds checking in memory, to provide comprehensive and efficient ROP defence without depending on information hiding.

Hardware-supported memory checks offer several advantages over software-based memory checking, including improved efficiency. However, we discovered that hardware extensions like Intel MPX must be carefully applied to fully harness the performance gains offered by such specialized hardware. In particular, not all operations supported by Intel MPX exhibit low overhead. Thus, our LMP design focuses on minimizing the utilization of high-overhead instructions and components of MPX while still effectively safeguarding critical memory regions from unauthorized modifications. In order to address the limitations of existing solutions and provide a more robust defense against ROP attacks, we have developed a prototype that showcases the capabilities of LMP.

By repurposing Intel MPX hardware extensions, our innovative approach enables fast and complete ROP protection without the need for information hiding. Our prototype demonstrates the ability to defeat ROP attacks while maintaining an average runtime overhead of only 3.9%.

## 1.2 Scalable Applications in Trusted Execution Environment

The rapid proliferation of mobile devices and their increasing integration into various aspects of our daily lives have amplified concerns surrounding mobile security. To address these concerns, hardware-assisted features such as ARM TrustZone have been employed to place mobile applications within a TEE to be isolated from the outside software environment. The TEE ensures that sensitive data and operations are protected from potential threats originating from untrusted parts of the system, including malicious software applications and corrupted operating systems that cannot be trusted. However, third-party applications are generally restricted from entering the TEE and benefiting from TrustZone's security features without first being vetted by the device manufacturers. One reason for this restriction is that current TEE operating systems are not designed to load untrusted applications and properly isolate them to support multiple mutually distrusting applications.

We propose a novel mechanism implemented as a TEE OS called Pearl-TEE, for functionality-specific isolation in TEE OS design, aiming to enhance the flexibility of TrustZone by allowing third-party applications to use the security features in TrustZone, while each application does not have to trust each other. We demonstrate the practical application of our OS design principles using mobile payment services as an example, showing how these principles can help democratize access to TrustZone for third-party applications.

Furthermore, to solve the problem of limited hardware monotonic counter resources in TEE, we proposed a new system Mint-TEE that leverages hardware features to enable an arbitrary number of TEE applications to allocate virtual counters at their discretion. This approach empowers these applications to defend against rollback attacks even when they only have the resource of a limited number of hardware monotonic counters.

To validate our proposed software solutions with hardware assistance, we have implemented two light-weight secure TEE OS prototypes: Pearl-TEE and Mint-TEE. These TEE OSes embody our design principles and provide robust security features for mobile services. Experimental results demonstrate that our system is both effective and efficient in delivering secure services while maintaining a high level of performance.

## 1.3 Hardware Features for Cache-based Side-Channel Attacks and Defence

Intel SGX was introduced as a hardware extension to provide a trusted execution environment for applications to run without having to trust the operating system that mediates between the application and the hardware. This hardware approach aimed to enhance the security of sensitive operations and data by isolating them from potentially compromised system components. However, despite its promising potential, the intended design of SGX functions does not entirely eliminate security threats. Specifically, it fails to protect secure applications running within the SGX enclave from cache-based side-channel attacks, which can expose sensitive information.

In response to this vulnerability, software developers have devised various defense mechanisms, including the creation of software high-resolution timers. These timers provide a means of mitigating side-channel attacks, despite the absence of hardware timers within the SGX enclaves. However, our work reveals that these software-based defenses are still susceptible to more sophisticated attacks.

We introduce a novel CPU thermal attack in our Aion attacks that repurposes processor features originally intended for managing CPU temperature. By exploiting these features, our attack alters the execution speed of the timer thread beyond what the defence can tolerate, undermining the efficacy of software-based defenses. Additionally, we present a cache eviction attack that targets timer counters, forcing the system to load them from memory instead of the cache. This further compromises the integrity of software high-resolution timers and exposes the limitations of current defenses against side-channel attacks in SGX enclaves.

In our research, we thoroughly evaluate the effectiveness of these Aion attacks and introduce an analytical model to better understand their implications. Our findings demonstrate that software timers, in their current form, cannot be sufficiently improved to protect against the threats posed by our attacks. This underscores the need for a more robust and comprehensive approach to securing SGX enclaves against cache-based side-channel attacks.

## 1.4 Contributions

We make the following contributions in this thesis:

- We repurposed the Intel MPX hardware extension to defend against return-oriented programming attacks, by developing a lightweight memory protection system. This system is designed to safeguard critical memory regions containing return addresses of function call stacks, particularly the shadow stacks. Our approach efficiently ensures that only legitimate accesses to the protected region are allowed, effectively preventing attackers from manipulating return addresses.

- We designed and implemented two TEE OSes for ARM TrustZone to support third-party, mutual distrusting applications to run securely in TrustZone, and to provide isolation, freshness and liveness to the TEE applications by creating unlimited number of virtual monotonic counters on top of the intended limited hardware monotonic counters.

- We proposed two types of generic Aion attacks with the assistance of existing hardware features to effectively exploit all existing SGX software timers which were assumed to be reliable for side-channel defenders. The attack model with existing hardware features shows that one cannot purely rely on a defence made with software timers and that new hardware may be necessary.

## 1.5 Thesis Outline

In this thesis, we begin in Chapter 3 by introducing our proposed solution to ROP attacks by using the MPX hardware extension. We describe the background of the security threats, explain the design of our LMP system, which consists of a customized compiler and a runtime library. We evaluate the performance overhead, code expansion and memory overhead of the LMP system.

In Chapter 4, we review background for on-board hardware and TEE application model before we propose our TrustZone repurposing TEE OS designs. Implementation details and evaluations follow to show the feasibility and effectiveness of the two TEE OSes.

Finally in Chapter 5, we show the framework of Aion attacks with the help of hardware features for manipulating software timers in SGX, allowing side-channel attackers to evade detection. Two types of attacks can work independently or in combination to make it more effective in practice. Background information is provided from Chapter 2, and we conclude this thesis in Chapter 6.

# Chapter 2

# Background

In this background chapter, we provide information about how hardware assistance features are emerged and used by the software developers for security-related purposes. We show the Intel's MPX for memory bound protection, and ARM's TrustZone for trusted execution environment along with other ARM on-board hardware. We also present a background on the cache-based side-channel attacks and defence on Intel's SGX.

## 2.1 Intel Memory Protection Extension

The hardware assistance, integrated within Intel's 6th Generation SkyLake processors [127], is known as MPX. Intel initially proposed MPX as a set of extensions to the x86-64 instruction set architecture, with the primary objective of monitoring pointer references at runtime, thereby mitigating illicit memory access incidents. The conception and development of MPX are rooted in the previously released Pointer Checker feature [53], implemented in the Intel compiler for debugging purposes. This feature generates a pair of bounds concomitantly with the creation of a pointer, and the compiler subsequently produces code to scrutinize these bounds when the pointer is utilized. While Pointer Checker relies solely on software, MPX facilitates hardware acceleration for bounds checks, which would have otherwise been conducted through software. Consequently, MPX encompasses both software and hardware components.

The hardware component of MPX incorporates several new registers and instructions into the instruction set architecture:

- 4 bound registers: `BND0` – `BND3`. Each register is 128-bit, comprising a 64-bit lower bound and a 64-bit upper bound.
- 2 configuration registers: `BNDCFGU` designated for user mode, and `IA32_BNDCFGS` for supervisor mode.
- 1 status register: `BNDSTATUS`, which retains error codes upon exception occurrences.

- Bound management instructions: BNDLDX and BNDSTX facilitate the loading of BND registers from a memory-based table containing object-specific address bounds. BNDMK and BNDMOV enable programmers to manually manage the BND registers.

- Bound check instructions: BNDCU and BNDCL ascertain that a pointer adheres to the upper and lower bound constraints specified by a particular BND register.

Regarding the software aspect, it necessitates support from system software components:

- MPX-enabled Compiler: The compiler is used for identifying when and where to generate and verify bounds by incorporating the relevant instructions. Bound values are also computed by the compiler through data flow and pointer analysis. Considering the potentially vast array of objects in memory, the size and bounds of all objects are retained in memory. Prior to a pointer accessing an object, the compiler loads one of the BND registers with the corresponding upper and lower bounds, subsequently utilizing the BNDCU and BNDCL instructions to verify these bounds. As of now, Intel has incorporated MPX support into the GCC main branch since version 5.0, catering exclusively to C/C++ and x86 targets.

- MPX Runtime: The MPX runtime library is linked to the program at compile-time. Executing at the user level, the runtime assists in configuring MPX hardware features, receiving and reporting errors, and managing the bound directory and bound tables stored in memory.

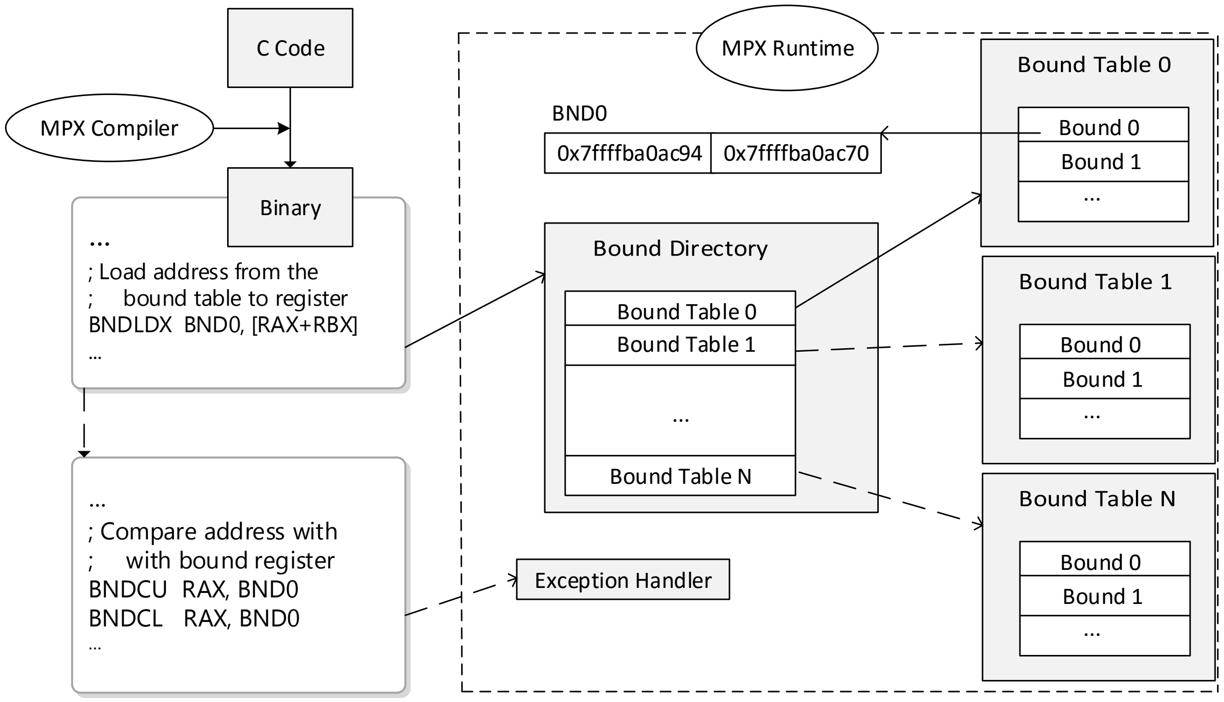
- Operating System: The operating system, in conjunction with the compiler, is required to accommodate the new instructions specific to MPX. In the event of a bound check instruction failure, the processor generates an exception due to a bound violation, necessitating the OS to handle the exception and signal the application accordingly.

We now present an example demonstrating how MPX components can be employed to perform bound-checking within a small program. Consider a program that declares and manipulates data across five arrays:

$$int \quad A[10],\, B[20],\, C[30],\, D[40],\, E[50];$$

Whenever a pointer pointing to one of these arrays is dereferenced, the MPX compiler must insert bound-checks to ensure that the pointer is situated within the bounds of the respective array. To accomplish this, the MPX compiler must first identify the intended target array for the pointer, subsequently loading the array's upper and lower bounds into a BND register. The appropriate BNDCU and BNDCL checks are then inserted prior to the pointer dereference, verifying the pointer's position relative to the upper and lower bounds of the array.

For instance, as depicted in Figure 2.1, if array A is stored at addresses 0x7ffffba0ac70 – 0x7ffffba0ac94, the MPX compiler must first load the upper and lower bound addresses 0x7ffffba0ac70 and 0x7ffffba0ac94 into one of the bound registers (e.g., BND0). This process is executed using the BNDLDX instruction, which transfers bound information from the bound directory in memory to the corresponding register. Subsequently, the MPX compiler instruments bound checking instructions to compare the pointer dereference with bound values stored in BND0. Should the dereference fall outside the specified bounds, a #BR exception will be generated by the hardware and intercepted by the exception handler within the MPX runtime.



```
...
; Load address from the
;     bound table to register
BNDLDX  BND0, [RAX+RBX]
...
```

```
...
; Compare address with
;     with bound register
BNDCU  RAX, BND0
BNDCL   RAX, BND0
...
```

Figure 2.1: An example for illustrating how MPX works to protect pointer bounds.

For a pointer referencing an array to undergo bound-checking, the bounds for that array must be loaded into a `BND` register. Given that arrays `A`, `B`, `C`, `D`, and `E` are all situated in distinct memory regions, the MPX compiler is required to load the appropriate array bounds into a `BND` register every time a pointer dereferences a location within a different array. As there are only four `BND` registers available and five arrays to consider, it is impossible for the compiler to maintain the bounds for all arrays in a `BND` register at all times. This scenario results in numerous `BNDLDX` and `BNDSTX` instructions being generated by the compiler to load and spill bounds information to and from memory.

The bound checking instructions (`BNDCU` and `BNDCL`) have a relatively low execution cost. However, the `BNDSTX` and `BNDLDX` instructions require access to the two-layer structured bound tables stored in main memory, rendering them considerably slower in comparison to the bound checking instructions. In order to measure this cost, we conducted an experiment comparing `BNDCU` with `BNDSTX`/`BNDLDX` instructions. We randomly generated 1000 memory addresses and used an address lower than all of them to perform 1000 `BNDCU` instructions, ensuring no bound violations occurred. Subsequently, we employed `BNDSTX` to store the first 500 instructions into bound tables and proceeded to load them all back one by one into a `BND0` bound register. The results of this experiment revealed that the bound checking instruction `BNDCU` exhibited an execution time nearly equivalent to a `NOP` instruction (1000 instructions in 0.45ms), while the bound store+load instructions `BNDSTX`/`BNDLDX` incurred a cost nearly 1000$\times$ greater than `NOP` (1000 instructions in 432ms).

In real-world applications, the number of objects within the bound table can grow substantially. However, since the number of `BND` registers remains fixed at four within the hardware architecture, this results in heavy reliance on BNDSTX and BNDLDX instructions, leading to considerable overhead. To demonstrate this in practice, we utilized a recent MPX-enabled version of GCC (version 6.1) to compile the SPEC 2006 benchmarks and discovered that this imposed a $2\times$ to $4\times$ runtime overhead. Consequently, these findings indicate that minimizing the number of BNDSTX and BNDLDX instructions is essential for maintaining low overhead. This optimization is one of the primary reasons LMP is capable of providing low overhead.

## 2.2 ARM Trusted Execution Environment and Mobile Secure Payment

ARM TrustZone is designed to be an open architecture for developing software in a trusted execution environment [125]. An ARM SoC with TrustZone partitions software and hardware resources, such as memory and access to peripherals into a secure and normal world, with the secure world isolated from the normal world with hardware protections. In the secure world, a small secure TEE OS provides basic security services, which enables secure applications to run in a TEE. Software running in the TEE can communicate with a normal-world OS, such as Android, and software in the normal-world using TrustZone drivers [59]. The GlobalPlatform API [56] is a common standard for such communication to take place.

The benefit of this division between a general-purpose normal-world OS and a small, TEE OS in the secure-world is that applications running in the TEE have a smaller trusted computing base (TCB), isolated from the large and complex normal-world OS. The rationale of the small and isolated environment of TCB is for easier analysis and verification, reduction of attack surfaces and minimizing risks of vulnerabilities. To maintain these benefits, the TEE OS must be high-assurance, meaning that it has passed a higher standard of code auditing and testing during its development.

To reach this higher standard, TEE OSes must be smaller, simpler and contain fewer features than general-purpose OSes. In the current ecosystem, only trusted, signed applications can run code on the TEE OSes. As a result, one of the areas where current TEE OSes reduce functionality in favor of simplicity is in the isolation they provide between TEE applications. This design choice leads to the current situation, where all mobile payment service applications that run in the TEE are either made by the phone manufacturer (e.g., Samsung Pay on Samsung phones [149] and Huawei Pay on Huawei Phones [34]), or by entities that are part of an industry alliance with the phone manufacturers (e.g., Internet Finance Authentication Alliance [79]). As a result, the security offered by TrustZone exclusively benefits some parties, while remaining out of reach for others. The excluded parties contain some very significant providers, such as PayPal, Alibaba and Tencent, who process billions of dollars in payments annually. These services and their many customers are not able to benefit from TrustZone's higher security assurance.

To the best of our knowledge, among all current commercial closed TEE OSs and open-sourced TEE OS (e.g., Google's Trusty TEE [60] and Linaro's OP-TEE [105]), there is no mechanism that can support third-party payment services to be installed, properly guarantee their integrity, and isolate the TEE applications so they do not affect other TEE applications. If they need to be designed to support third-party payment systems, mechanisms like secure attestations and runtime memory isolation shall be required. Our research addresses the domain of payment systems, but allows for isolation of untrusted applications in the TEE for specific payment functions.

## 2.3 Other ARM On-Board Hardware Related to Security

Based on the commonly available hardware and development of mobile device, we introduce the following hardware on board, which we will mention them in as part of the system we describe in the thesis:

- **Secure Element**. To attest the integrity of software running on a device to a remote party, a trusted hardware device in the form of a Trusted Platform Module (TPM) or a Secure Element (SE) is required. Both of these are security-hardened processing elements capable of securely storing signing keys and performing cryptographic operations with those keys. Since the keys are never exposed to software running on the application processor, signatures made by these keys can be trusted even if the remote party does not trust the integrity of the normal-world OS.

- **Touch Device**. This module can directly generate a signal when a real user touches it. The signal, later transmitted directly via TEE OS, proves to the remote payment server that the payment is intended by a real person. A trusted path from the touch device to the TEE software is required to ensure that a malicious normal-world OS cannot simulate a touch without the cooperation of the user.

- **Trusted Display**. The TEE software needs a way to securely communicate with the user to display payment details so that the user can confirm them. Without this channel, malicious software in the normal-world could display one set of payment details (such as amount or payee) and submit a different set to the TEE software. We note that our implementation of the payment flow, described later in this chapter, does not need security indicators as in other work [164, 103], but instead relies on the user verifying that the payment details displayed are consistent with their intentions.

## 2.4 Intel SGX, TSX and Cache Structures

In this section, we describe hardware background required by understanding cache-based side-channel attacks and defence in main chapters of this thesis. Intel SGX [84, 31] provides a trusted execution environment which is targeted by some of the attackers, and a few proposed research projects use Intel Transactional Synchronization Extensions (TSX) [83] for their defensive mechanisms, which we will also introduce later. Then, we will give an overview of the cache structures in Intel processors as the cache is a commonly used structure that many side-channel attacks exploit.

### 2.4.1 Intel SGX

Intel SGX is an instruction set extension introduced in 2015 to the Intel architecture. SGX offers facilities designed for security and system properties, including: Confidentiality, where memory content is encrypted by the CPU and inaccessible to unauthorized parties; Isolation, enabling the CPU to support iso lated execution of SGX applications separate from non-SGX applications; Integrity, allowing the memory content to be verified for integrity through SGX hardware features; and Verifiability, providing a remote infrastructure for attestation that confirms the platform of the executed secure application is genuine. SGX establishes a TEE for user-level applications to securely operate within a protected environment called an enclave. Security properties are ensured by placing application code and data into Processor Reserved Memory (PRM), which is isolated from the main memory and encrypted transparently by the Memory Encryption Engine (MEE). Secure applications in enclaves can be interrupted by other applications outside the enclave, resulting in an Asynchronous Enclave Exit (AEX) event.

The trust model of SGX allows enclave applications to operate without trusting the underlying OS or hypervisor. However, this also imposes restrictions on the applications, such as prohibiting syscalls to the underlying OS and disallowing certain instructions inside an SGX enclave such as hardware clock counter `RDTSCP` instructions and performance monitor unit instructions.

### 2.4.2 Intel TSX

Intel TSX [83] is an instruction set extension that provides hardware transactional memory support. TSX ensures that when a sequence of instructions is executed, either the execution is completed without interruption or memory read-write conflict (i.e., concurrent access to the same data where at least one access is a write) with other threads, or the transaction is aborted, and the execution is rolled back. The most significant distinction from other transactional memory schemes is that TSX is implemented by hardware, so it does not depend on any software locks, thereby reducing most of the overheads.

Originally, TSX was intended to accelerate multi-threaded applications by reducing locking, rather than enhancing software security. However, recent works have utilized TSX as an auxiliary method for notifying the secure enclave that is supported by Intel SGX about interruptions by other threads [156] or for protecting cryptographic keys against memory disclosure attacks [66].

TSX-supported compilers provide new intrinsic functions for software to safeguard critical regions. Enclave protection code leveraging TSX uses the intrinsic function calls to mark the beginning and end of critical code regions. When the code executes, if there are any other threads attempting to access the same memory region accessed by the critical code, a transaction abort will occur, regardless of the threads' privilege levels, and the memory will roll back to the point of transaction entrance. The thread under TSX protection will then be informed about the interrupts, and the enclave protection code can count the number of interrupt occurrences.

### 2.4.3 Intel Cache Structures

The side-channel attacks and defence issues we are discussing in this thesis are all related to the cache channels, thus we introduce some common background information about the cache structures of current Intel processors. Cache is a widely used layer in the memory hierarchy designed to reduce traffic and improve latency when serving memory requests. Typically situated transparently between processors and the main memory (i.e., managed by the hardware), cache is organized hierarchically, with level 1 (L1) cache closest to processors and Last Level Cache (LLC) closest to the main memory. In modern Intel processors, each CPU core (which can run two logical threads concurrently) has a split L1 data and L1 instruction cache and a unified L2 cache. Additionally, there is usually a sliced L3 (last level) cache shared among all CPU cores, where each memory address is mapped to a specific cache slice through an unpublished mapping function [115], and each cache slice can independently serve memory requests.

Within each hierarchical level, cache organizes data in units of cache lines, which are the smallest units for cache filling and eviction. Cache is typically set-associative, meaning that it is logically divided into equally sized cache sets, and each memory address is mapped to one of the cache sets. An N-way (set-)associative cache indicates that each cache set has the capacity for N cache lines. If the cache is sliced, like the LLC in Intel CPUs, all ways of the same cache set will be in the same cache slice. When a memory request arrives, the cache uses a middle part of the requested address (set index) to find the cache set, then compares the higher address bits (address tag) with that of each cache line in the set to locate the correct line, and finally uses the lower address bits (line offset) to extract the requested portion of the cache line. Cache replacement (or eviction) policy refers to the strategy for choosing a cache line to evict when a cache set is already full but a new line needs to be added. One example is the LRU (least recently used) policy, which always evicts the cache line that has not been accessed for the longest amount of time.

For efficiency and implementation reasons, hardware-managed multi-level caches often constrain how the same data may appear in different cache hierarchy levels [69]. Inclusive caches require lower-level caches to include all data from upper-level caches, while exclusive caches prevent the same data from being cached in more than one level. Although some recent designs feature exclusive caches, most Intel CPUs still use an inclusive cache design, where eviction from the LLC guarantees eviction from upper-level caches.

## 2.5 Power and Thermal Management of Intel CPU

Modern CPUs have power and thermal management features designed to maintain energy efficiency and protect the processors physically. For example, in Intel CPU thermal management [84], each package of the CPU contains several digital thermal sensors (DTS) for temperature detection, and the results can be retrieved from MSR registers or PCI interfaces. For safety concerns, such as CPUs running too hot, the thermal control circuit (TCC) is responsible for responding when a certain temperature limit is reached: it may either reduce the CPU frequencies, the voltages on processors, or force the CPU to go into duty cycling.

The TCC can be activated under circumstances configured by system administrators with privilege. If the TCC offset is misconfigured, thermal events could be triggered at a low threshold, meaning CPU performance would be throttled down earlier than necessary. Such an event can also be triggered by software, trapping the CPU into a mode where processors reduce their power consumption by using clock modulation.

To the best of our knowledge, there is currently no established attack that utilizes the feature of TCC activation for malicious purposes. However, we point out that if this feature is maliciously used, CPU speed can be a target that is easily manipulated without being detected by threads running on the controlled core. This potential vulnerability will be further discussed in this thesis.

# Chapter 3

# LMP: Light-Weighted Memory Protection with Hardware Assistance

## 3.1 Motivation of the LMP System

In this section, we first introduce the ROP attack and current mitigation techniques against ROP attacks as the main motivation of developing a new memory protection mechanism. Then, we briefly describe what the defence we propose is and what the contributions it makes.

### 3.1.1 Attack and Defence of ROP

In languages such as C/C++, the programmers are ultimately responsible for enforcing the memory safety of their programs. However, inevitably, programmers produce code with flaws that violate memory safety, and some of these flaws result in memory corruption vulnerabilities that allow attackers to maliciously alter the control flow of programs [145, 120], corrupt critical data [73, 111], or cause sensitive information leakage [44, 184].

There have been numerous proposed or deployed defenses to mitigate memory corruption vulnerabilities [177]. Despite this, memory corruption vulnerabilities continue to be exploitable. For example, ASLR (Address Space Layout Randomization) [134] randomizes memory locations of code and data segments, but can be circumvented via vulnerabilities such as address space leakage, timing side-channels [78] or attacks such as just-in-time code reuse [159]. DEP (Data Execution Prevention) [4] prevents injecting and executing new code in vulnerable programs. However, it cannot prevent reusing existing code in an application via a return-to-libc or ROP (Return-Oriented Programming) attack [145].

To address ROP attacks, Abadi et al. propose Control-Flow Integrity (CFI) [1]. CFI protection enforces both forward-edge protection (i.e. indirect function calls) and backwards-edge protection (i.e. function returns) to ensure that a memory corruption vulnerability does not allow an attacker to corrupt a code pointer and redirect execution along an edge not specified by the original program. While the target of a forward-edge function call can be resolved to a single or small number of targets statically, the target of a backwards-edge function return cannot generally be determined with much precision using only static analysis. As a result, backwards-edge protection generally requires a runtime component. To determine and enforce backward-edges precisely, shadow stacks are proposed in the first CFI paper [1] as a separate stack used in conjunction with the regular call stack, to store return addresses of function calls in a program. WHen a function is called, the return address is pushed onto both the regular call stack and the shadow stack. When the function returns, the return address is popped from the shadow stack, and its value is compared to the one on the regular call stack. If the two addresses mathc, the control flow is deemed legitimate and execution proceeds as normal, otherwise indicating a possible tampering with the call stack happening, which makes it more difficult for an attacker to manipulate a program's control flow. Then, the mechanism proposed in software-based fault isolation (SFI) [176] is further used to protect the contents of the shadow stacks from corruption by an attacker. Unfortunately, the runtime overhead of the memory checking required to properly implement this runtime component can be as high as 2× [33].

To reduce this overhead, various proposals weaken the properties of the backwards-edge protection in return for better runtime performance. For example, some propose coarse-grain protections, which do not use a shadow stack to precisely track backwards-edge targets. Since shadow stacks are not used, there is no need for SFI, which avoids the expensive checks required to implement memory protection for the shadow stacks. This coarse-grain approach is taken by proposals such as kBouncer [132], ROPGuard [52], ROPecker [24], which have significantly lower overheads ranging from 1.59% to 2.60%. These coarse-grain methods are imprecise in that they do not actually validate that the return address on a backwards-edge actually points to the original caller; instead, they either only check that the return address points to an instruction that follows some call instruction, or they heuristically check the number of returns to detect gadgets executions. They have all been shown to be circumventable [38, 58] and ineffective against a knowledgeable attacker.

Information hiding is another way to mitigate the overhead of complete CFI backwards-edge protection. In this approach, rather than protecting the data in the shadow stacks with memory access checks, the shadow stacks are placed at a random location in a 64-bit address space. Because the size of the address space is large, it is assumed infeasible for the attacker to guess the location of the shadow stacks. One method called code-pointer integrity (CPI) [101] is able to provide CFI protection with 2.9% overhead (on C applications). However, information hiding techniques can be broken by memory safety vulnerabilities that leak the location of the shadow stacks [49]. Other work has also shown that various side-channel attacks can be used to leak information that can be used to find the hidden shadow stacks [153, 163]. The lesson here is that ultimately information hiding is not equivalent to memory protection, as they are vulnerable to address information leakage, while memory protection is not.

### 3.1.2 Contributions of the LMP System

In this chapter, we propose Light-Weighted Memory Protection (LMP) [74], a new method that leverages Intel's Memory Protection Extensions (MPX) to make backwards-edge CFI both secure and efficient. LMP tackles two essential problems that stand in the way of memory safety in system software: critical memory region protection in backwards-edge CFI approaches and nontrivial overheads in checking memory access violations.

While hardware-supported memory checks are naturally more efficient than software memory checking, which is also proven in recent work on using customized hardware for CFI enforcement [27, 37], we find that the hardware extensions like Intel MPX have to be applied carefully in order to truly reap the performance benefits of specialized hardware. In particular, not all of the operations supported by Intel MPX have low overhead. Therefore, we design LMP to minimize the use of the high-overhead components of MPX and still enable it to effectively protect shadow stacks from unauthorized modification.

We build a proof-of-concept prototype implementation of LMP and measure the performance overhead with SPEC 2006 benchmarks. The LMP system introduces an average overhead of 3.90%, which is much less than the 2× overhead from the reference implementation of the original CFI [33]. In fact, LMP achieves roughly same overhead as information hiding techniques [101, 36], which have generally about 3% overhead. LMP is also comparable with recent coarse-grained CFI approaches, which have overheads between 1.59% (ROPGuard [52]) and 2.60% (kBouncer [132]). However, LMP provides stronger security guarantees than both information hiding and coarse-grain approaches, as it is both not vulnerable to either side-channel leakage and enforces a much stricter policy.

We summarize three main contributions the chapter of LMP makes:

1. We propose an alternative use of hardware assisted pointer checker with Intel MPX that is different from the standard proposed use of MPX.

2. We provide the first stack protection solution that is assisted by the available CPU feature of Intel MPX.

3. We achieve a low overhead among existing equivalent solutions, while provide stronger protection than coarse-grain backward-edge CFI approaches.

## 3.2 Design of the LMP System

### 3.2.1 Threat Model

We assume a realistic attacker capable of exploiting memory corruption vulnerabilities to modify content of arbitrary memory addresses. We also assume that the attacker is aware of the address locations of key data structures, such as pointers, stacks, and metadata, and can arbitrarily target them using memory corruption vulnerabilities. The attacker's goal is to corrupt a code pointer to compromise the control-flow integrity of a program.

Despite this powerful attacker, we do assume some realistic limitations. For instance, the attacker cannot directly modify CPU registers or alter any memory marked as read-only, such as code pages, as both actions would enable the attacker to remove or bypass the compiler-inserted instrumentation used by LMP. Additionally, attackers cannot compromise the target program's integrity before it is loaded into memory, meaning that attacks on the program loader and operating system are out of scope for LMP. LMP aims to mitigate the exploitation of memory corruption vulnerabilities by remote or unprivileged attackers for the purpose of privilege escalation.

In general, two types of code pointers require protection: function pointers (i.e., forward-edge) and return addresses (i.e., backward-edge). LMP focuses on defending against attacks on return addresses and assumes the use of an existing forward-edge CFI protection scheme to safeguard function pointers from corruption, and such an implementation of forward CFI can guarantee forward-edge protection along with LMP. There is a rich body of literature addressing forward-edge protection. For example, virtual calls in C++ indirect-control transfers through VTables can be hijacked by attackers [20] to redirect execution to malicious code. Low-overhead protections for such cases have been developed in previous works, such as VTV [168], VTable Interleaving [13], and VTrust [203]. Our LMP system can work in conjunction with current forward-edge CFI defenses to provide comprehensive CFI protection.

### 3.2.2 LMP Memory Protection with the assistance of MPX

LMP employs two components to protect return addresses: shadow stacks and the protected memory region allocator. First, standard shadow stacks are used to maintain a second copy of return addresses. The shadow stack is updated on a function call and checked when functions return. To successfully corrupt a return address, an attacker would have to corrupt both the program stack at the function call site and the shadow stack. Therefore, to prevent the attacker from corrupting the shadow stack, LMP inserts MPX instructions to ensure that only the instructions it adds at function calls to update the shadow stack can write to the shadow stack.

According to the threat model delineated in Section 4.2.1, only store operations possess the capacity to modify the memory region of the shadow stack. As the code pages maintain a read-only status, an attacker is incapable of eliminating bound checks related to store operations. An attacker may attempt to bypass the bound-checks by directly jumping to a store instruction. However, this necessitates corrupting a code pointer, which the Control Flow Integrity (CFI) supplied by LMP, in conjunction with a supplementary forward-edge CFI scheme, effectively prevents. Consequently, the backward-edge protection offered by LMP depends on the capability to safeguard shadow stacks against corruption resulting from memory safety vulnerabilities.

To defend the shadow stack, each store instruction within the program is instrumented to ensure inaccessibility to the memory region of shadow stacks, even if the attacker alters the targeted instruction's effective address. Despite the presence of numerous store instructions, all are subject to the same bounds, as LMP merely verifies that they do not target the shadow stack. This approach is efficient, as it circumvents the necessity of employing costly `BNDLDX` and `BNDSTX` operations to modify the bounds that LMP examines. LMP simply establishes the upper and lower bounds of a `BND` register corresponding to the shadow stack's lower and upper regions, then proceeds to instrument each store within the program to guarantee it remains outside of that region. However, in the context of multi-threaded programs, each thread contains a unique shadow stack. A simplistic solution would involve utilizing a separate `BND` register to store the upper and lower addresses for each stack, but this would mandate the use of the high-cost `BNDLDX` and `BNDSTX` instructions to load and store the stack bounds into the `BND` registers, negatively impacting performance. Alternatively, we note that all shadow stacks belong to the same protection class, signifying that irrespective of the executing thread, a store should be incapable of accessing any shadow stack. Consequently, all shadow stacks can be positioned within a contiguous memory region and protected using a singular `BND` register. Thus, another component of LMP encompasses a scheme that allocates standard shadow stacks within a single contiguous memory region. Similarly, all other auxiliary data structures employed by LMP are protected from modification by being allocated within the protected region, which is

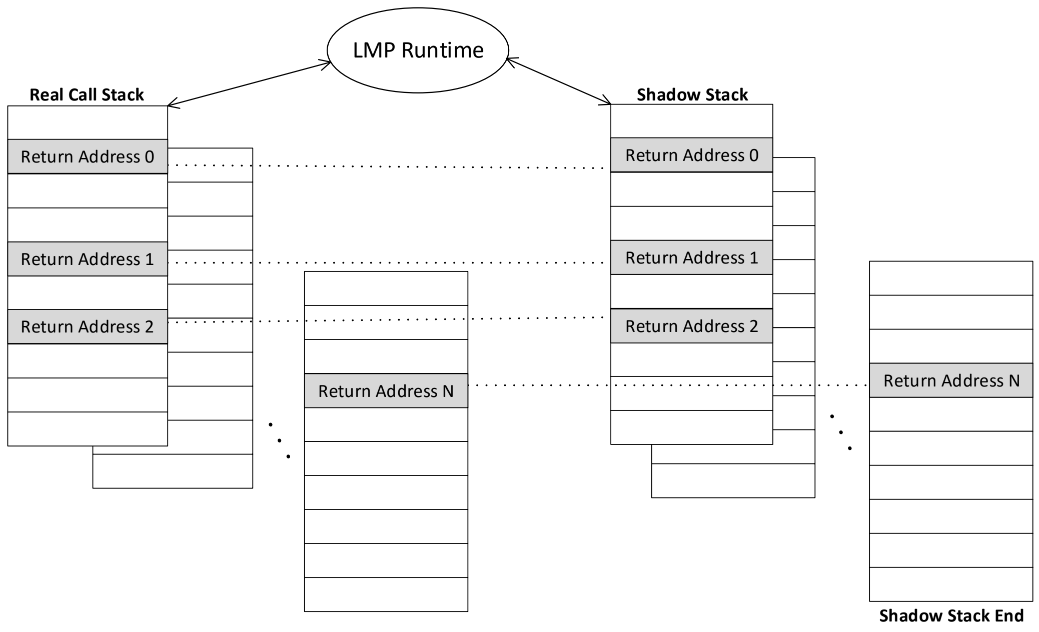


Figure 3.1: The illustration of LMP shadow stacks

restricted by MPX instructions.

### 3.2.3 Using the Shadow Stack

In order to restrict return instructions, LMP records the return address in the shadow stack upon each function call, where it will be protected from corruption by an attacker. We illustrate the idea of shadow stack layout of the LMP system in Figure 3.1.

In order to constrain return instructions, LMP records the return address in the shadow stack upon each function call, effectively safeguarding it from potential corruption by an attacker. The conceptualization of the shadow stack layout within the LMP system is depicted in Figure 3.1.

A further distinction from alternative shadow stack approaches lies in LMP's comparison of function return addresses with those stored in the shadow stack utilizing MPX bound checking instructions. This method optimizes the overhead associated with compare/branch instructions in standard shadow stack implementations, the details of which will be discussed later in this section.

As previously noted, shadow stacks reside in a contiguous memory region. Additionally, this region is statically defined at program initiation, and due to its inaccessibility to any memory instruction other than shadow stack operations implemented by LMP, the region cannot be employed to store any data type other than shadow stacks. A primary contrast between our shadow stack implementation and other shadow stack or safe stack implementations [36] is that LMP is not supposed to freely place shadow stacks in any location or offset-based region, but must instead place them to the predefined shadow stack region. Given that each thread must have its own shadow stack, we must define a mapping function that allows the shadow stack code to find the shadow stack for any given thread, but also maps each shadow stack into the predefined memory region.

One potential solution involves designating the predefined region to be as extensive as the area where regular stacks can be allocated. This approach would be efficient, as each shadow stack could then be situated at a fixed offset from the thread's regular stack. However, the pthread interface allows for stacks to be created anywhere within a process' virtual address space. Consequently, one half of the virtual address space would have to be reserved for the predefined region. While this may be acceptable in the majority of cases for 64-bit code, it can pose challenges if processes require memory allocation at a specific virtual address space.

Alternatively, a more resource-intensive but adaptable option is to dynamically allocate and map stack space from the predefined region as threads and their corresponding shadow stacks are created. Although this may be marginally more costly than the fixed-offset method, it remains practical and offers a more conservative estimation of the overhead attributed to various LMP implementation choices. LMP employs a mapping table that records the offset between a thread's regular stack and its corresponding shadow stack. Both the function entry and function return instrumentation utilize the mapping table to locate the appropriate shadow stack for the thread. The predefined region is subsequently divided into several fixed-sized shadow stacks, with an additional table storing which shadow stacks are in use and which are available. Upon thread creation, LMP identifies an unallocated shadow stack and updates the mapping table with the offset between the thread's regular stack and its newly assigned shadow stack. When a thread is terminated, the thread is deallocated and the offset in the table is removed. These allocation and de-allocation procedures occur exclusively during thread creation and destruction.

LMP integrates instrumentation at function entry, storing the return address into the shadow stack. As this memory operation is incorporated by LMP, bound-checking is unnecessary. At function return, LMP inserts instrumentation that locates the corresponding return address in the shadow stack and compares it against the address targeted by the control flow. In this manner, the shadow stack verifies that the integrity of the return address remains unaltered upon execution return. A thread's regular and shadow stack share an identical layout, which ensures that a return address on the regular stack has the same offset from the base of the stack as the corresponding return address' offset from the shadow stack's base. Therefore, only the offset between the regular stack base and the shadow stack base must be stored in the mapping table. This design is different from RAD [26] which uses a custom stack layout, and because of this, it has to search through the shadow stack to for match-finding, while LMP does not need to.

An example of the execution sequence following code instrumentation for shadow stack operations is provided, along with an assembly code snippet in Figure 3.2:

1. On function entry:
   (1) prepare shadow stack address in register %rax
   (2) copy return address in %rsp to shadow stack
2. Execute function call and body
3. On function return:
   (1) copy return address in shadow stack to bound register %bnd0
   (2) use bound checking instruction to check return address in %rsp and %bnd0

```
…
PUSH   %rsp
CALL   _map_table                  # find shadow stack via mapping table
                                   # return shadow stack address in %rax
…
MOV    (%rsp), %rdx
MOV    %rdx, (%rax)                # copy ret addr to shadow stack address in %rax
...
(FUNCTION CALL BODY)
...
MOV     (%rsp), %rdx               # put function return address in %rdx
BNDMK %bnd0, [(%rax), 0]           # put the address in shadow stack in a bnd
                                   # register %bnd0
BNDCU  %rdx, %bnd0
BNDCL  %rdx, %bnd0                 # check return address with the one in shadow
                                   # stack
...
```

Figure 3.2: Assembly code example for instrumented function entry/exit.

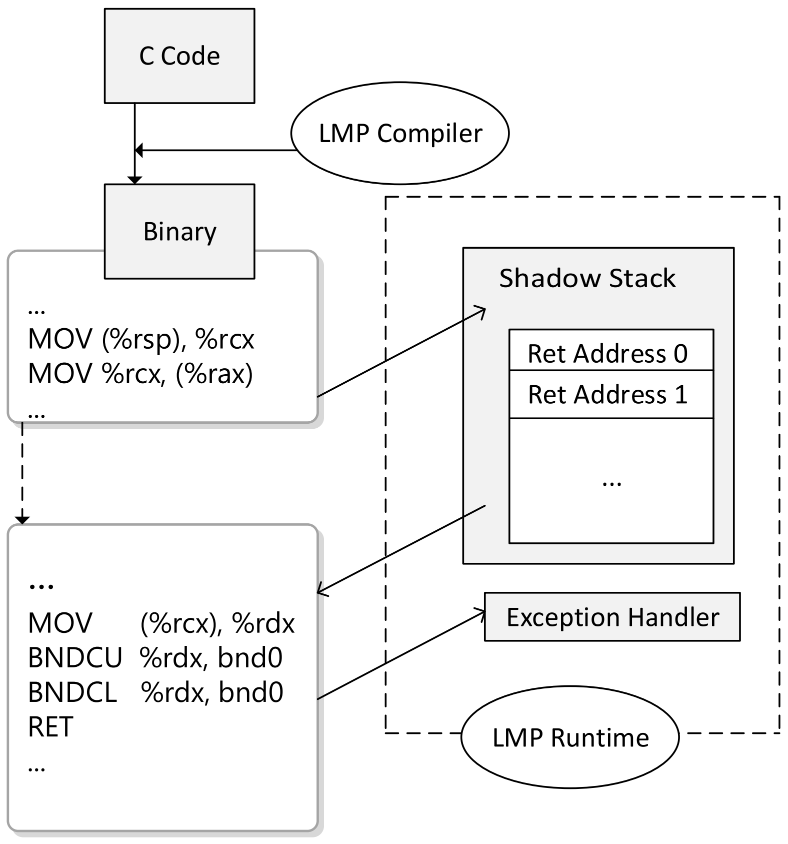
MPX bound checking instructions `BNDCL` and `BNDCU` are employed instead of a series of compare and jump instructions for equality comparison. The return address in the shadow stack is set as the upper and lower bound in the bound register (`BND0`), which is then bound-checked against the function return address. Utilizing MPX instructions to verify the return address enhances performance in the same manner as MPX instructions improve memory bound-checks – these MPX instructions circumvent additional branch and check instructions typically required to verify the comparison result. Instead, if the check fails, MPX instructions will generate an exception.

### 3.2.4 Execute a Program with LMP

An illustration of the conceptual design of our LMP system is provided through a simple example demonstrating the interaction of LMP with a user program, as depicted in Figure 3.3.

The LMP-enabled compiler instruments the application source code during compile-time. At program initiation, the LMP runtime configures the shadow stack memory region and records its lower boundary and upper boundary in the bound register `BND1`. This action serves to protect the shadow stack from unauthorized modification. As the program runs, return addresses are stored in the shadow stack when a function call occurs and the return address is pushed onto the standard call stack. Upon the function's return, the two addresses stored in the normal stack and shadow stack are compared. Throughout the program, any memory operation that stores values to a memory address is instrumented to ensure that the address is not within the range of the shadow stack using bound checking instructions.

In specific exceptional cases, such as C++ exception handling, the call stack may unwind due to `setjmp`/`longjmp` instructions, resulting in function call and return mismatches. In the LMP method proposed, as long as the compiler does not alter the original call stack with exception information



```
...
MOV (%rsp), %rcx
MOV %rcx, (%rax)
...
```

```
...
MOV      (%rcx), %rdx
BNDCU  %rdx, bnd0
BNDCL   %rdx, bnd0
RET
...
```

Figure 3.3: A flow chart of how LMP system works.

(e.g., GCC stores it in an additional side-table), the return addresses in the original call stacks and shadow stacks correspond to the same offset to the stack top addresses, ensuring that stack unwinding due to exception handling operations remains unaffected.

Although it has not been implemented, it is posited that LMP can be extended to offer backward-edge protection for binary-only CFI. With a control-flow graph (CFG) generated through disassembly analysis of a binary and modifications to pthread library functions, LMP can also be compatible with binary-only CFI approaches that utilize binary-rewriting to integrate CFI instrumentation.

## 3.3 Implementation of the LMP System

The LMP comprises two primary components: the LMP-enabled compiler and the LMP runtime library. The compiler portion entails modifying the register transfer language (RTL) passes for instrumenting boundary checking, ensuring that no unauthorized writes can occur within the memory region storing the shadow stacks. The LMP runtime is responsible for managing shadow stack allocation and the storage of return addresses from function call stacks.

### 3.3.1 LMP-enabled Compiler

The LMP-enabled compiler implementation is based on GCC 5.2.0, with approximately 600 lines of code modified or added to the RTL passes. The primary rationale for altering the compiler and incorporating new RTL passes is to enable code instrumentation at the assembly level. Both shadow stack operations and code designed to protect the shadow stack memory region from modification are instrumented by the LMP compiler.

Within the GCC RTL passes, the source code in `final.c` and `insnoutput.c`, which handle assembler code output for functions, is modified. `final_end_function()` aids in emitting assembly code at function exit, and instrumentation for shadow stack operations is added here.

To implement shadow stacks, at each function call stack operation when the function pushes the return address, the compiler instruments the code to acquire the address and save a copy to the thread's shadow stack. The shadow stack location is determined by indexing into the stack region using the calling thread's Thread ID, which is obtained via the system call `gettid()`. Although it may initially appear that a system call would be excessively costly, such operations are highly optimized, and measurements indicate that the expense of `gettid()` on modern Linux kernels is negligible. At each return instruction, the compiler instruments the code to obtain the Thread ID and request the return address stored in the shadow stack from the LMP runtime. If the address in the return instruction does not match the one in the shadow stack, a bound violation message is sent to the LMP runtime. In the GCC passes, function calls are identified by searching for the RTL expression code `call_insn`, formatted as follows:

$$(call \quad (mem : fm \quad addr) \quad nbytes)$$

where the `addr` represents the address of the relevant subroutine.

For bound checking of memory operations, the RTL passes of GCC are modified to locate RTL expressions containing memory operations that store values to the primary memory address. The

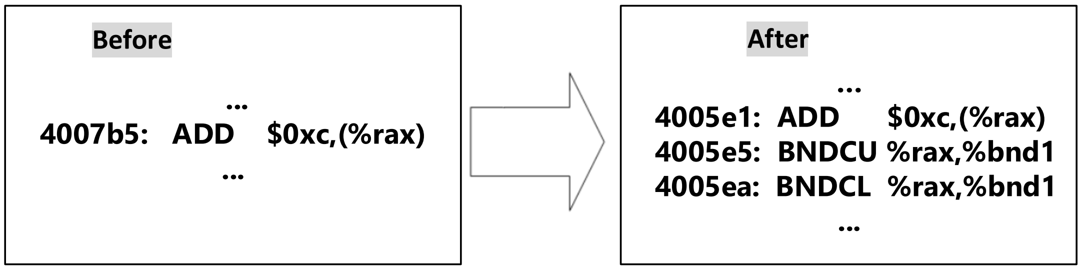


Figure 3.4: An example of LMP instrumentation for store instruction.

address is compared with the upper and lower boundary addresses of the shadow stack, which are stored in the bound register `BND1`, housing the bounds of the memory region where the shadow stacks reside. A bound violation will be triggered if the address falls within the memory range of the shadow stack, indicating that the pointer used as a target by the memory store may have been compromised by an attacker.

An example of code instrumentation results is presented in Figure. 3.4, where `(%rax)` is an access to memory using `%rax` as a pointer. The figure displays the assembly code before and after instrumentation. The `add` instruction writes to the main memory, while the instrumented assembly code, `BNDCU` and `BNDCL`, verifies whether the memory address to be modified falls within the protected shadow stack region, the checks can also be inserted before memory operations.

## 3.3.2 LMP Runtime

The LMP runtime is implemented using approximately 700 lines of C source code. As a proof-of-concept prototype design, a virtual memory region of 2GB is allocated for the shadow stacks. This memory size is chosen based on our test environment's maximum of 62,057 threads (from `/proc/sys/kernel/threads-max`), allocating 32KB to the shadow stack for each potential thread. This allocation is deemed sufficient, as the benchmarks employed never exceed 8KB per thread in the call stack. In our implementation, both the maximum number of threads and the space allocated for each shadow stack are adjustable. Since the shadow stacks are allocated within the 64-bit virtual address space, they occupy only a small fraction of it. Moreover, as most shadow stacks may never be written to, they consume merely virtual address space, and the operating system is not required to allocate physical memory to support them.

Alternatively, dynamic allocation of shadow stacks in memory could have been employed, allowing the shadow stack region to be dynamically extended and reduced in size to accommodate growth and reduction in shadow stack usage. While this approach might add some overhead in exchange for better virtual address space utilization, it is not deemed necessary given that virtual address space is generally not a limiting factor on 64-bit architectures.

When the instrumented program requires the LMP runtime to store a function return address in the shadow stack, the runtime calculates the offset between the base of the call stack and the address storing the return address, as well as a Thread ID to process them in the function `LMP_push_ss(return_addr, offset, threadID)`. The runtime then locates the shadow stack designated for that thread and stores the function return address in the shadow stack. When the program function returns and the address must be compared with the one stored in the shadow stack, the offset between the base of the call stack and the address storing the function return address is calculated, utilizing `return_addr=LMP_pop_ss(offset, threadID)`. Subsequently, the LMP runtime retrieves the return address stored in the shadow stack.

## 3.4 Evaluation of the LMP System

In this section we evaluate the effectiveness and different aspects of overheads of our LMP system. We run our experiments on an Intel i5-6600K with 4 cores @3.5GHz in 64-bit mode with 8G RAM. The benchmarks are run on Fedora 22 with Linux kernel 4.1.7.

### 3.4.1 Measurement of Performance Overhead

The overheads of the LMP system are assessed using the CINT 2006 benchmarks. All results represent the average of five runs obtained from the non-reportable mode of the SPEC benchmark. These results are compared with the baseline performance without the implementation of LMP. As demonstrated in Figure. 3.5, the average performance overhead of LMP compared to the baseline is 3.90%. The `h264ref` benchmark exhibits the highest overhead at 12.55%, primarily due to its significantly greater number of function calls and `RET` instructions compared to other benchmarks. Excluding the `h264ref` benchmark, the average overhead is reduced to 2.12%.

The relatively low overhead after instrumentations on memory operations is benefitted mainly by MPX's low-cost bound checking instructions. As mentioned in Chapter 2.1 with our measurement, the bound checking instructions like `BNDCU` and `BNDCL` cost nearly as low as a `NOP` instruction when the bound is stored in a bound register like `BND0`. Also, the instruction execution related to bound checking is possibly boosted in CPU instruction parallelism as other studies show [127], noting that even `NOP`s are not free either. Therefore we brought low overhead with LMP by not relying on the slow bound load and store instructions, but only using fast bound checking instructions provided by MPX.

To identify the primary sources of overhead introduced by the LMP system, the overhead is further divided into three components: context settings, bound-checking, and shadow stack operations. Context settings encompass runtime library initialization, ThreadID retrieval via system calls, and other related processes. Bound-checking accounts for the time consumed by MPX bound instructions, while shadow stack operations consist of all processes associated with the shadow stacks.

Each component's contribution to the overall overhead is measured by removing the other two components and assessing the overhead with only one component added to each benchmark. Across all CINT 2006 benchmark results, the average overhead of context settings is 0.1%, bound-checking is 0.52%, and shadow stack operations is 3.27%. As illustrated in Figure 3.6, context setting and bound-checking contribute negligible overhead. Shadow stack operations are the main contributor, accounting for an average of 84% of the total overhead. The performance penalty of memory protection constitutes only 15% of the overall overhead, with the remaining 1% attributable to infrequent setup and stack allocation/de-allocation operations. These results are consistent with other heavily optimized shadow stack implementations [36] that report performance overheads between 2%

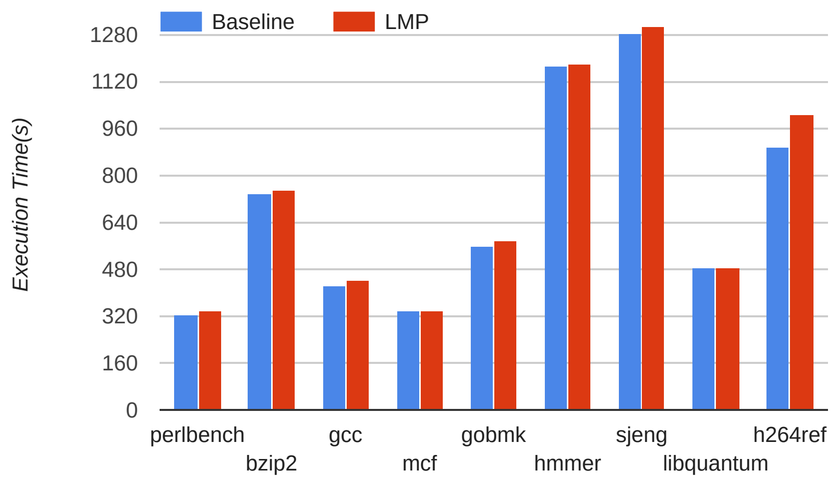


Figure 3.5: LMP overhead by comparison of execution time between baseline and LMP.

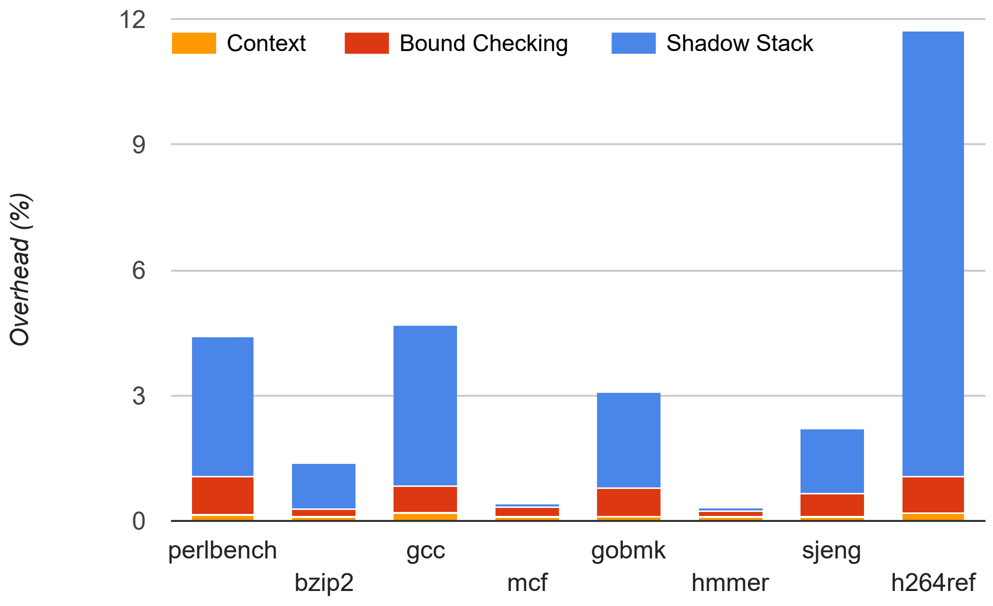


Figure 3.6: Overhead components of LMP.

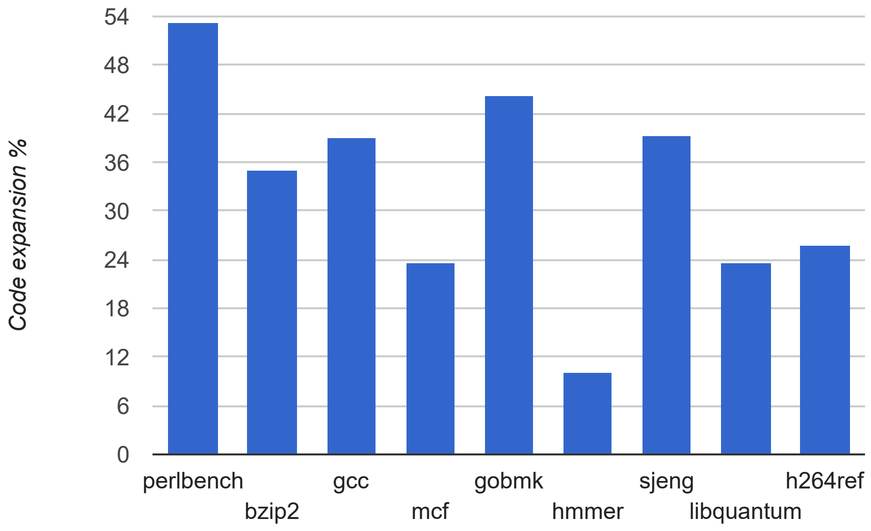


Figure 3.7: Code Expansion of LMP.

and 10% for the same benchmark set. Consequently, it is reasonable to consider this overhead as representative of the costs of LMP on current processors.

### 3.4.2 Code Expansion Evaluation

The LMP-enabled GCC generates assembly code to instrument the target program during the RTL passes, resulting in an increase in code size. A direct comparison of the binary sizes for each benchmark is performed to calculate the percentage of code expansion introduced by LMP.

As depicted in Figure. 3.7, the assembly-level code expands by an average of 39.27% across the nine benchmarks executed. There is some variability in the code expansion percentages among the benchmarks, with the majority of the expansion resulting from bound-checking instructions. When a benchmark features more function calls/returns and memory store instructions, additional bound-checking instructions are instrumented. It should be noted that the prototype includes extra code originally for the purpose of debugging the program, which could be removed to further reduce code expansion of LMP.

### 3.4.3 Memory Overhead Comparison

The memory overhead introduced to the benchmarks is, on average, 19.3 MB per program, and the average percentage of the maximum resident memory overhead is 9.73%. The memory overhead mainly originates from the LMP system's runtime library, which manages the shadow stacks. As discussed in Sec. 3.3, the memory allocation in this research prototype implementation is not optimized, and it is believed that there is still potential for further improvement on performance. By incorporating dynamic allocation of the mapping table as needed, instead of preemptively allocating it for the maximum number of threads, the memory overhead could be significantly reduced.

## 3.5 Related Work

This section reviews the literature on defense technologies designed to protect programs from control flow hijacking attacks, including proposed hardware-assisted methods, and discussion on new hardware security extensions.

Hardware-assisted control-flow integrity protections have emerged across different platforms. Camouflage [40] and PAL [201] apply ARMv8.3 pointer authentication extension for protections on Linux kernel with fine-grained CFI approaches. CEST [200] and CFI CaRE [126] use ARM TrustZone-M to ensure CFI on embedded systems for both user and kernel programs. New hardware extensions have been proposed for RSIC-V architecture and ISA in order to enforce CFI, for example, Escouteloup [48] discuss adding ISA support for enforcing CFI by using frame pointer marked as confidential in registers. FIXER [39] proposes using a co-processor for protecting the program control flows. While this chapter focuses on using Intel hardware extensions, there have been other works adopting different extensions such as GRIFFIN [54] and PT-CFI [65] use Intel Processor Trace extension to enforce or verify program control flows.

Traditional attack methods, such as stack-smashing and code injection [138], can be defended against using data execution prevention (DEP) [4]. Hardware support for DEP is present in virtually all x86 processors as a non-execute bit (NX bit, also referred to as XD/XN bit depending on the processor architecture), preventing code in the data segment from being executed. In response to these protections, attackers have developed more sophisticated methods that do not rely on injecting new code and instead exploit existing code in the program. Early examples include the return-into-libc attack [170], which redirects program execution flow through `Libc` functions. Similarly, return-oriented programming (ROP) attacks [145] execute arbitrary computations by utilizing a chain of existing code after altering the return address in the function call stack. ROP attacks have been demonstrated to be Turing-complete.

Randomization is practical in hiding information about the memory layout of a program from attackers. Address Space Layout Randomization (ASLR) [134] has been proposed to defend against ROP attacks by mapping program processes and dynamic libraries into random virtual address space every time. Address Space Layout Permutation (ASLP) further re-orders sub-routines at the code segments on the basis of the randomization provided by ASLR [93]. However, the implementations of ASLR were soon found to be ineffective against de-randomization attacks [154], requiring only a few hundred additional seconds to compromise the target program. ASLP is also vulnerable to de-randomization attacks [110].

Control Flow Integrity [1] is introduced to guarantee that indirect control-flow transfers point to legitimate locations. To ensure that the return addresses in function call stacks are not tampered with, shadow stacks to store copies of return addresses are suggested. However, the performance overhead of original CFI is reported as high as 2$\times$ if the exact policy is enforced, so there are variants of coarse-grained CFI proposed with changes to the original policy. kBouncer [132] utilizes the Last Branch Record (LBR) x86 register that stores recent branches executed by the CPU. It validates whether the return address points to an instruction following a call instruction, effectively serving as a heuristic mitigation of ROP attacks. Employing the same LBR register and a similar policy as kBouncer, ROPecker [24] incorporates additional static analysis to speculate on a program's future execution in order to defend against ROP gadgets. However, this approach can also be bypassed [38]. ROPGuard [52] checks if the stack pointer points to a memory address outside of the stack, preventing ROP attackers from executing payloads on the heap. Nevertheless, adversaries can still modify the stack pointer before the target function is called. The aforementioned defenses are also vulnerable to attacks that leverage hooks and hide malicious code within non-control data [175] if the critical memory region is not protected at runtime. O-CFI [121] explores a randomization method to conceal the program's control-flow graph and applies MPX in bound-checking for guarding branch instructions. However, it is still a coarse-grained CFI method and provides only probabilistic security guarantees, as it does not fully protect function return addresses. Our LMP approach adheres to the original CFI policy for backward-edge protection, i.e., checking every function return address and ensuring the return address points to the function caller.

Forward-edge CFI protection is not the main focus of this chapter, thus we only briefly discuss it here as it is related to backward-edge CFI. The paper proposing VTV [168] reports that more than 90% of indirect calls are virtual calls. Their method focuses on protecting VTables from being hijacked, validating at runtime that the target VTables are part of a legitimate set before a virtual method call is made. The performance of VTV depends on the size of the legitimate VTable set, so the complexity of the C++ class hierarchy affects the overhead. Building on this idea, VTrust [203] and VTable Interleaving [13] improve the performance of VTV without requiring global class hierarchy and prevent VTable hijacking attacks. Our LMP system does not provide protection with forward-edge CFI. However, the LMP can be easily combined with the aforementioned approaches by applying patches to the LMP-enabled compiler, thus enabling full-CFI protection.

There are CFI variants proposed with different security targets. The techniques of original CFI have been used for the purpose of enforcing software-based fault isolation (SFI) [202]. XFI [47] also employs CFI policies with the help of debugging information in Windows PDB files to defend against ROP attacks. Data-flow Integrity (DFI) [19] follows CFI approach to prevent non-control data attacks. Hypersafe [183] is similar to fine-grained CFI protection. It has a target table for indirect branches and aims at protecting control-flow integrity of hypervisor.

Code-Pointer Integrity (CPI) [101] explores a security mechanism that divides process memory into two parts: safe memory region and regular memory region. Through static analysis, memory objects that have pointers including code and data pointers are put into a safe memory region for protection against illegal tampering. However, flaws of CPI approach have been pointed out [49] because its safe memory region is not well-protected. The essential idea of LMP is also guarding the memory region where shadow stacks located. We use new hardware feature of fast memory boundary checking to ensure the allocated shadow stack region is protected effectively and efficiently.

Other hardware-based CFI approaches have been proposed, such as HCFI [27] and HAFIX [37] which have their systems implemented on customized FPGA boards or SPARC embedded systems. Hardware-based memory safety defenses have also been proposed such as In-Fat Pointer [195] that achieve spatial memory safety at subobject granularity. In contrast, LMP is the first system with hardware-assisted memory protection compatible with commercially available CPUs and other hardware. Other repurposing usage of Intel MPX have also emerged such as for secure storage [29] and side-channel protection [99].

Control-Flow Enforcement Technology (CET) [81] was announced in 2016. CET introduces a new exception class (#CP) with interrupt vector 21 and a new `ENDBRANCH` ISA instruction to help mark legal targets for an indirect branch or jump. It also uses hardware protections to limit access to the shadow stack, allowing only function call and return instructions to modify the shadow stack and prohibiting regular memory stores from doing so. Works like CETIS [193] also repurposes the use of Intel CET from shadow stack protection to intra-process memory isolation. Our LMP system could not be feasible after Intel's MPX official deprecation in 2019, leaving Intel CET the closest solution for protecting the shadow stack with hardware assistance. However, as recently emerging new attacks pose challenges to Intel CET's protection with CFI, for example, the Untangle attack [11] and the WarpAttack [194], the hardware-assisted security on memory protection still has a long way to go.

## 3.6 Summary and Discussions

Memory protection serves as a cornerstone for all defense techniques against memory corruption attacks. Without adequately safeguarding the shadow stack, CFI approaches cannot effectively thwart ROP attackers and have generally been proven insecure. Our study proposes a lightweight memory protection system designed to prevent unauthorized access to critical memory regions that store return addresses of function call stacks, specifically the shadow stacks. By utilizing the Intel MPX hardware features, our approach achieves low overhead in ensuring only legitimate accesses to the protected region are permitted, thus preventing attackers from tampering with return addresses. As future work, we plan to extend LMP protection to forward-edge control flow and explore the potential for applying LMP without the constraint of recompiling the program. For instance, we may employ binary rewriting to implement shadow stack functions for enhanced protection.

The lesson we learned from this chapter reveal that by making well-considered adjustments to the original use cases put forth by the architects of hardware security extensions, it becomes feasible to attain a satisfactory level of security while significantly reducing system overhead. The example of LMP serves as an illustration of this principle, demonstrating that although achieving comprehensive security protections may be challenging and resource-intensive, there are viable alternative approaches that can offer a satisfactory level of protection at a significantly lower cost. Security hardware can be deprecated, removed from support, and discontinued eventually, as the case study of Intel MPX shows in this chapter, however, the example of repurposing of Intel MPX provides a pathway towards pragmatic solutions that convert existing tools with heavy overheads to a new light-weight approach.

Chapter 4

# Repurposing ARM TrustZone for Third-Party Applications and Roll-Back Protection

ARM TrustZone technology has emerged as a hardware-based security solution, aimed at protecting devices from various security threats, including tampering and unauthorized access by malicious applications or operating systems. This technology enables device manufacturers to provide secure execution environments for sensitive applications and data through the isolation of code execution from the underlying commodity operating system. Despite its numerous advantages, the use of TrustZone has been largely restricted to device manufacturers, limiting the ability of third-party application developers to leverage its security features. In this chapter, we propose a novel approach to make TrustZone more accessible and open to third-party developers, allowing them to benefit from its security capabilities.

To achieve this objective, we have designed and developed two prototype TrustZone operating systems, Pearl-TEE [76] and Mint-TEE [75], that demonstrate the feasibility of using TrustZone technology for secure third-party application development and for developing security mechanisms to defend against rollback attacks. These prototype OSes showcase how the technology can be adapted to accommodate a variety of security properties, addressing the needs of different applications and use cases. Our research provides a comprehensive analysis of the design and implementation of these prototype systems, detailing the challenges faced during development and the solutions employed to overcome them.

## 4.1 Motivation of Repurposing ARM TrustZone

### 4.1.1 Current ARM TrustZone Use

ARM TrustZone is a powerful mechanism for reducing the attack surface of security-sensitive applications on the ARM platform. TrustZone implements a hardware-based TEE, which reduces the attack surface of an application by enabling it to execute code that is isolated and protected from other, lower-assurance code on the system, such as the general-purpose operating system and applications. Instead, TrustZone applications execute in an isolated environment with their own TEE OSes. This isolation provides strong confidentiality and integrity guarantees for applications executing in the TEE and the data processed by the TEE applications. TrustZone is currently being used in smartphone operating systems such as Android to protect a small handful of applications that implement services such as mobile payment and DRM.

The existence of such a useful security mechanism might beg the question: Why is TrustZone not used by security-sensitive applications in general? Indeed, neither iOS nor Android exposes a public API that enables application code to access TrustZone. While strategic business exclusivity may be one reason for this omission, a concrete technical reason is that small size and simplicity are required to maintain the high-assurance level of the TEE OS. Because of this, current TEE OSes are unable to isolate TEE applications (i.e., applications running in TrustZone) from each other. Instead, every application that runs in the TEE must trust every other application, and so the only way to ensure the trustworthiness of all TEE applications is to limit which applications can run in the TEE. As a result, only a small number of TEE applications that are signed and pre-installed by the smartphone vendor are allowed to benefit from TrustZone.

However, in recent years, there have increasingly been efforts to make TrustZone and other processor-assisted TEEs to be more open, to both developers and end users. For example, Apple allows password manager software to use the Secure Element on iOS devices [5], Trustonic released an open TrustZone operating system framework called Kinibi [171] and Google is working on an open-source TEE OS called Trusty [60], which aims to have an interface that will allow 3rd party applications to execute code in TrustZone.

Supporting a larger number and variety of TEE applications (TEE apps) creates conflicting design goals —on one hand it is desirable to allow as many applications as possible to benefit from TrustZone enabling them to run application components in the TEE, but on the other hand, creating an OS for the TEE that can handle many, mutually distrustful applications, will also increase the complexity of the TEE OS and thus decrease assurance. In the worst case, if the TEE OS must support every application a commodity OS does, then it will likely be similar in size and complexity (as well as assurance) to a commodity OS, and thus any benefit of the TEE will be lost. In this work, we have investigated the common uses of TEE apps and come up with a TEE application model as a guidance of what TEE OS is supposed to support.

### 4.1.2 TEE Application Model and Examples

Before describing our abstract TEE application model, we begin by describing several common classes of applications that currently use, or may benefit from, the security provided by TrustZone.

**Payment Applications** Malware targeting mobile payment applications has become more prevalent [167, 165, 169]; thus, some phone manufacturers now use TrustZone to isolate their payment applications from malware and vulnerabilities in the general-purpose OS. To do this, mobile applications need to authenticate the user and transmit and receive confidential payment information. Furthermore, they execute within the TEE to ensure that their operations are protected from malware in the general-purpose OS. Given that they must sign operations as well as verify the authenticity of responses sent to them, they must store long-term cryptographic keys securely. Examples of payment applications that use TrustZone include HuaweiPay [34] and SamsungPay [149]; on the other hand, PayPal [135] and AliPay [3] are examples of applications that don't.

**Digital Right Management** Digital Right Management (DRM) systems intend to control the use, copy and distribution of copyrighted software and data. Unlike payment applications, DRM systems do not take user input from the trusted touch screen, but only require a secure path to output video content. DRM applications also need to authenticate users to ensure that they are authorized to view content, or to use a particular piece of software. They also need to communicate with remote servers to verify the content that the user is authorized to access. Finally, DRM systems must protect and use keys to decrypt content so that it becomes available for the user to view or use. For example, XBox and PlayStation encrypt game-related data so that they cannot be copied across multiple devices [50]. Widevine, a DRM application commonly deployed on smartphones, uses TrustZone to protect its secrets and secure its execution [188].

**Authentication** Authentication applications are used to provide the second factor in two-factor authentication in the form of a pseudorandom sequence of secrets that are generated by the device. The pseudorandom sequence is seeded by a secret and the server and device are synchronized so that the server knows what indexes in the sequence it might receive. As a result, such an application must be able to store secrets and operate on them in a way that is protected from tampering or observation. When initializing the application, the remote server typically needs to authenticate the user so it can synchronize its index with that of the particular user. Many software two-factor authentication applications, such as Google Authenticator and Microsoft Authenticator, exist, but as far as we know, none currently use TrustZone to protect their execution.

**Integrity Monitoring** Integrity monitoring applications continuously monitor normal-world applications and the general-purpose OS to ensure that their execution patterns imply that the software has not been compromised. To ensure integrity, some integrity monitoring applications, such as Samsung Knox [90] may run in TrustZone. While the operation of Samsung Knox is not public, we surmise that they execute in TrustZone to protect their integrity against a possibly compromised general-purpose OS, and also possibly to protect cryptographic secrets with which they may securely communicate the state of their device to a remote server for mobile device management (MDM) functionality.

**Secure Chat Application** Many mobile chat applications provide secure end-to-end chat with message encryption, e.g., Telegram and Xabber. Chat applications share similar properties with payment applications: they communicate with an external server via messages, require secure key storage for encryption keys, and need to run in isolation from one another. Furthermore, to ensure that users see correct messages, they require a secure channel to the screen. Finally, secure chat applications need to verify user identity to the remote server and/or end-user. Currently, we are not aware of any secure chat applications that can use TrustZone.

Despite the security benefits of using TrustZone, many applications cannot benefit from TrustZone because they cannot run on existing TEE OSes. Both current open-source [105, 60, 158] and closed-source [147] TEE OSes do not isolate TEE applications from each other, nor do they isolate themselves from the TEE applications, meaning that all TEE applications installed on a user device must mutually trust each other. Due to this limitation, device manufacturers simply disallow installation of any new TEE applications, and only run those TEE applications that come pre-installed on the device. Recent academic work has proposed the use of a trusted language runtime for multiple applications in the TEE [150]; however, the trusted runtime does not provide a level of isolation that would prevent a malicious application from compromising another co-resident application. This

Table 4.1: Traits of each class of TEE applications

| Application class | Key storage | Integrity /confidentiality | Secure comm with server | Secure comm with user | Identity provision |
|---|---|---|---|---|---|
| **Payment** | ✓ | ✓ | ✓ | ✓ | ✓ |
| **DRM** | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Authentication** | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Integrity Monitor** | ✓ | ✓ | ✓ | | |
| **Secure Chat** | ✓ | ✓ | ✓ | ✓ | ✓ |

trust requirement severely limits the accessibility and availability of TrustZone to applications, as a single party must be trusted by all applications in the TEE to curate and vet all the code in the TEE. Moreover, all TEE applications must come installed on the phone and be updated simultaneously with a firmware update. Installing individual TEE applications on the fly is still not possible in the trusted language runtime the way it is, for example, possible for normal Android applications.

With this TEE application model, we can then propose our new TEE OSes that better serve the need of TEE apps and users with security guarantees.

### 4.1.3 Our Approach and Contributions

One approach to provide rich functionality in a TEE OS without the accompanying code bloat is to *borrow* functionality from the commodity OS in a way that does not compromise the security of TEE apps running in the TEE OS. This idea of using functionality in an untrusted commodity OS has been proposed in other work [8, 118, 23, 72, 9, 210]. For example, TEE apps can outsource file storage to the commodity OS. Protecting the confidentiality and integrity of data stored in the commodity OS can be accomplished with standard encryption and digital signatures. However, these mechanisms do not protect against a malicious commodity OS rolling back such files to an older state. Such an attack could have serious consequences—the file rolled back could cause an application to use expired certificates or stale credentials, which may put the TEE App's security at risk.

The generally accepted solution to prevent rollback attacks is to use either a hardware or a network-based monotonic counter to verify the freshness of the data stored in the commodity OS. However, each of these approaches has drawbacks. A network-based counter requires an online-challenge response to prevent the network messages themselves from being replayed, which may cause counter increments to incur latencies of several hundred milliseconds if made across a wide area network [114]. In addition, network counters require the TEE to trust the remote network counter. On the other hand, hardware counters are local and do not suffer from network latency, but suffer from limited durability—both the number of hardware counters and the number of times they can be incremented are limited and cannot be reset. For example, one of the current typical ARM SoCs has two hardware counters called "High Endurance Monotonic Counters" [117], each of which can be incremented $2,097,151$ times. Intel SGX [85] can support up to 256 monotonic counters which are implemented within Intel ME, and which will wear out after about 1 million writes [114]. Evidently, current implementations of hardware counters do not scale well to an increasing number of TEE Apps.

To solve the above issues, we propose two systems, to support untrusted applications in TrustZone and to assist scalable rollback protection for TrustZone applications, respectively:

1. Pearl-TEE, which shows that for TEE applications, the requirement of mutual trust among TEE applications can be eliminated with less than a 3% increase to the trusted computing base (TCB) size of the TEE OS. To do this, we first define an *abstract TEE application* model, which encompasses the types of applications that can benefit from TrustZone's security guarantees. We then implement a Pearl-TEE prototype TEE OS on top of an existing TEE OS to estimate the amount of extra code that needs to be added to support the abstract TEE application model. Finally, we demonstrate that our prototype is realistic by porting three mobile payment applications and one secure chat application to run in TrustZone.

2. Mint-TEE, which addresses the limitations of hardware counters by utilizing resources in the commodity OS to allow an arbitrary number of TEE Apps to create as many *virtual counters* as they need with latencies comparable to that of hardware monotonic counters. Mint-TEE also alleviates the memory pressure created from having to support a potentially large number of TEE Apps by using commodity OS memory as swap, while still protecting the confidentiality, integrity and freshness of swapped out TEE App state. Mint-TEE implements *aggregation* mechanisms that allow multiple virtual counter increments to be aggregated into a single hardware counter increment, thus reducing wear on the hardware counters, as well as allowing multiple hardware counters to be aggregated together to support an arbitrary number of virtual counters.

A challenge we faced with designing the TEE OSes is that many TEE applications, such as mobile payments or DRM, need to securely establish an identity for the user using the application. However, they cannot trust services in the general-purpose OS (such as Google Play Services) to assert the identity of the user via protocols such as OAuth. On the other hand, since TEE applications themselves are no longer trusted, having the user provide their username and password credentials to each TEE application that is installed exposes the user to the possibility of phishing attacks by fake TEE applications. To address this challenge, we design a novel *TEE identity attestation* mechanism, which leverages our TEE OSes' existing ability to attest the integrity of code, to also attest the identity of the user. To ensure that the identity of the user that the TEE OSes attest is genuine, we restrict this identity to only be changed when the device is in a pristine factory state, such as during device initialization. TEE identity attestation is the first proposal we are aware of to incorporate user authentication, as well as code attestation into a TEE OS. We make the following contributions in this chapter:

- We propose an abstract TEE application model and show how common TEE applications map onto that model

- We show our implementation of Pearl-TEE, a prototype TEE OS that satisfies the requirements of our abstract TEE application model.

- We elaborate how Mint-TEE provides isolation, freshness and liveness to TEE Apps and at the same time, borrows resources from the commodity OS to provide scalability for TEE Apps. Mint-TEE implements two forms of aggregation: *increment aggregation* that reduces hardware counters wear, and *counter aggregation* that allows multiple hardware counters to be virtualized together.

- We evaluate the increase to the TCB of modifying an existing TEE OS to implement Pearl-TEE and Mint-TEE, and find that they only increases the TCB by less than 3%. We evaluate both prototypes' performance and practicality by porting four existing applications and find that they impose less than 10% performance overhead.

## 4.2 Pearl-TEE: Democratizing TrustZone using Functionality-Specific Isolation

### 4.2.1 Threat Model

We first describe the threat model used in our design of Pearl-TEE. First, we assume that the user mobile devices have ARM TrustZone support or other equivalent hardware support for TEE, and listed on-board hardware. We also assume that the entire normal-world OS (e.g., Android and all applications on top of it) can be potentially compromised. The capabilities of the attackers include intercepting network communications, installing malicious applications, and gaining root permission on the normal-world OS. Furthermore, we assume the attackers can use malicious applications to impersonate benign TEE applications for phishing users' account information, such as passwords. In the secure-world, we assume untrusted applications can be loaded to TEE, and they can only run in the unprivileged level EL0. Finally, we make standard Dolev-Yao assumptions about the attacker's ability to attack cryptographic algorithms used in Pearl-TEE — namely that an attacker cannot recover plaintexts from ciphertexts without the key, and that the attacker cannot recover encryption keys from an arbitrary number of ciphertext/plaintext pairs.

In our threat model, we assume that the attackers do not have physical control of the device, and are thus unable to perform invasive attacks that modify the bootloader, perform cold-boot attacks, or use side-channels such as the last-level cache side-channel to implement their attacks. We assume that the touch sensor is able to authenticate the user via their fingerprint, or that, if it only registers touches, the user uses some strong authentication method to lock their phone. In this way, if their phone is stolen, an attacker cannot use the phone to access TEE applications using the user's identity. We also assume that devices do not come with malware installed and that the factory software that runs at device initialization (or after a reset) is trustworthy. Finally, we assume the presence of secure-boot so that certain persistent secrets are made available only to a trusted TEE OS binary.

Under the above assumptions, our TEE OS can ensure that a malicious party would not be able to: 1) steal and use the user's account to impersonate the user by another TEE application on the same or different device, 2) tamper with the operations of other TEE applications, such as changing the destination of a transaction, 3) affect or tamper with the operations of the TEE OS, for example, steal or change the master key of the TEE OS, and 4) if the malicious party is a TEE application it would not be able to affect normal operations of the normal-world OS.

We note that because TEE applications need to be initiated from the normal-world OS, and we assume that the normal-world OS may be compromised by an attacker, Pearl-TEE cannot guarantee the availability of TEE applications as a compromised normal-world OS can always refuse to run the TEE application to completion.

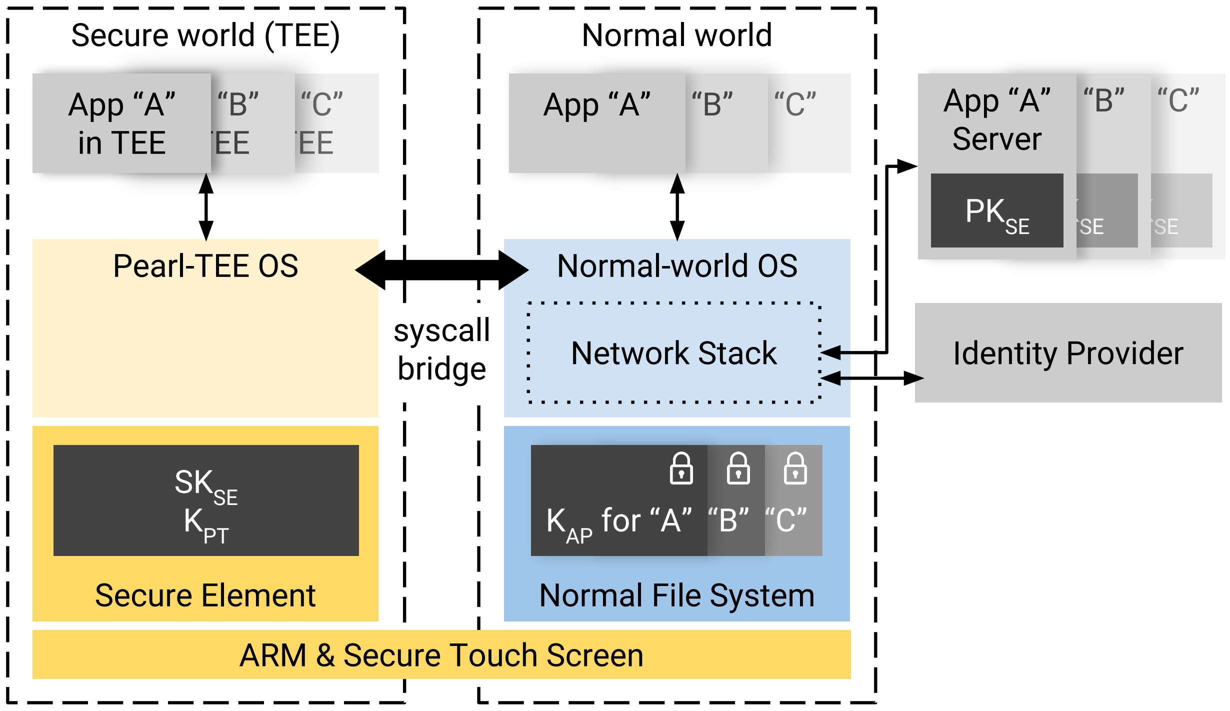


Figure 4.1: System overview of Pearl-TEE with multiple TEE-enabled applications.

### 4.2.2 Design Requirements

We survey current mobile payment applications to understand their security requirements in a single-tenant environment, where it is assumed that there is no malicious software on the system that could compromise the payment application. We then add requirements that arise in a multi-tenant environment where a payment application may run alongside malicious software intent on compromising the payment application.

#### Requirements for a Single-Tenant Payment Application

To consider a general case that only one payment application exists in the TrustZone that dedicates to serving one mobile payment service provider and assisting the user to complete payment transactions, we introduce this single-tenant abstract payment service model, which can be described as follows:

A payment server is run by the payment provider and takes requests from the user payment application. The user uses the payment application to send payment messages to the server that specify a) the identity of the payer, b) the payment amount and c) the identity of the payee. We envision an attacker who may also run applications on the phone that has some or all of the following goals:

1. The attacker may try to tamper with the payment message to change the identity of the payee/payer or payment amount.

2. The attacker may try to impersonate the user and use the application to send a payment the user does not intend.

To defeat these attacks, a single-tenant payment must provide the following guarantees:

- **S-1**: The payment application must execute in a protected environment and have secure storage for a secret signing key with which it can sign payment messages. Such a signing key fulfills two purposes. First, it is only available to legitimate payment applications, so it proves the integrity of the payment application that sent the payment message to the server. Second, each user has a unique signing key, so it also authenticates the user who used the payment application to send the message. A secure execution environment is required to prevent an attacker from tampering with the payment message while it is being constructed or tampering with the execution of the payment application. We use the term *payment message* to denote any request from the user that must be secured before sending to the payment server, which could be a command to transfer funds to another party, but could also be commands to confirm payments, cancel payments or view balances. While some payment applications rely on the integrity of the general-purpose OS on the device to provide a protected execution environment and secure storage, an attacker that is able to circumvent these could tamper with the payment application's execution or steal cryptographic keys. More secure payment applications (e.g. Samsung Pay) can execute within a TrustZone TEE. Current TEE OSs such as OP-TEE[105], TLK[123] and SierraTEE [158] use a chain of trust rooted in a secure boot process to assert the integrity and identity of both the TEE and code running within it to the remote server. In this way, the payment server can detect if the payment application or its execution environment has been tampered with or not. Furthermore, TEEs provide secure storage or a means of deriving a private key that cannot be accessed by code outside of the TEE.

- **S-2**: The payment application must verify that a payment request came from the user before issuing a payment message to the server, even if the payment application has not been tampered with. To do this, the payment application must authenticate a payment request from the user. First, the payment application needs a trusted path to the user so that it can display payment details, such as the payment amount and payee. Then, to ensure that the user approves of the payment request, the payment application waits for the user to confirm the request. This can be accomplished by having the user provide a credential, such as a password or a fingerprint to prove that the user is present and initiated the payment request. In addition, a trusted path between the user interface and the payment application in TEE must exist to ensure that user credentials are not stolen and replayed by a malicious application on the phone.

- **S-3**: The payment applications must be isolated from the TEE OS. Since TEE applications are not trusted, the TEE OS and payment applications should not run at the same exception level, so as to ensure that a malicious payment application could not tamper with the TEE OS or access sensitive data. Specifically, TEE OS runs at EL1 (exception level 1) while the payment applications run at EL0 (exception level 0). This enables the TEE OS to use hardware mechanisms such as the MMU to protect its code and execution state from tampering by a malicious payment application.

S-1 to S-3 can be satisfied by current TEE OSs. However, as will be shown next, current TEE OSs cannot meet the additional isolation requirements for multi-tenant operations.

**Requirements for Multi-Tenant Model for Payment Applications**

The previous scenario assumed either only a single payment application in the TEE or that all payment applications in the TEE mutually trusted each other. In the event of a multi-tenant payment application system, where the payment applications in the TEE do not trust each other, further requirements must be met by the TEE OS to ensure the security of payment applications.

- **M-1**: Storage isolation for signing keys to prevent unauthorized access to keys by malicious applications. To defend against attackers that try to impersonate a payment application, the TEE OS must ensure that keys used to sign payment messages are only accessible by the payment application that owns the key. As a consequence, this also requires a way for the TEE OS to establish the identity of a payment application so it can derive the key or set of keys that should be made available to that application.
- **M-2**: Isolation and integrity of the payment application code and execution. The code and execution state of a payment application must be protected by the TEE OS from tampering by other TEE applications. In addition, the TEE OS must guarantee transactional atomicity so that any payment operation must either complete or fail in a well-defined way that is understood consistently by both the user and the payment provider. Finally, the TEE OS must be able to prove the integrity of the application to the payment server.
- **M-3**: Fairness between payment applications controlled by the TEE OS to prevent denial of service attacks. While high performance and low latency is not a requirement, we do not want a single payment application to prevent other payment applications on the device from being used.

We point out that availability is not a guarantee that Pearl-TEE can provide, and UI/Phishing attacks are not types of threats we consider here. Because of the limited and simple nature of the code that is executed in TrustZone, all applications must necessarily have a component that runs in the normal world. This component can be subverted by a compromised OS, which can always prevent the user from making payment transactions by refusing to communicate with Pearl-TEE. As a result, the availability of payment transactions is explicitly not guaranteed, as this would require moving all the functionalities of payment applications into TrustZone, thus significantly increasing the TCB. Pearl-TEE is designed to satisfy these TEE OS requirements without increasing the TCB size too much. We describe the philosophy behind the FSI design and how it is applied to achieve this goal next.

### 4.2.3 Pearl-TEE Design

The additional requirements **M-1** to **M-3** all involve logical and/or resource isolation. However, we note that payment applications do not require the generic, high-performance isolation that commodity OSs provide. As a result, we use FSI mechanisms, which use simpler mechanisms than those used by commodity OSs, but still provide the necessary level of isolation to implement Pearl-TEE.

FSI design revolves around three principles. First, identify the specific requirements that the application needs in contrast to generic requirements that commodity systems provide. Second, design mechanisms that only provide the required functionality, making trade-offs in complexity for unrequired functionality. Finally, leverage existing functionality in the untrusted normal-world OS or underlying hardware when it is safe to do so to avoid bloating the TCB of the TEE.

We describe how we can apply these FSI design principles to meet requirements **M-1** to **M-3**:

**M-1: Key storage encryption.** Commodity OSs use a file system (FS) to provide applications with secure storage. An FS assumes that files may be written and read arbitrarily frequently, and may store a wide range of data sizes. The need to manage changing data further introduces the requirement of commodity FS to be fail-safe and preserve data in the event of a power failure, hardware fault, or software fault. In contrast, payment applications typically only need to store small keys, which are only written once, and are never shared. Because they do not change, there is little need for reliability as well, as the persistence of the underlying storage medium is sufficient. Thus, the complexity of a full FS is unnecessary for Pearl-TEE.

Pearl-TEE avoids this complexity by only providing a small amount of write-once and read-only storage for a payment application to store keys. Since the value of the keys does not change, Pearl-TEE does not need to guarantee freshness. Any tampering of the storage by an adversary will result in invalid keys, which will result in invalid signatures on payment messages. Thus, Pearl-TEE only needs to guarantee the confidentiality of the keys. This is achieved by encrypting all application keys with an application-specific storage key that only Pearl-TEE has access to. Pearl-TEE leverages the persistence provided by the normal-world FS and storing the encrypted keys in the normal-world FS. To ensure that a payment application only obtains the keys that belong to it, Pearl-TEE identifies a payment application by the hash of its binary and associates this with its keys. If a payment application's binary is changed or updated, the payment application must write new keys to Pearl-TEE's storage using the Initial Software Setup (ISS) procedure described in Section 4.2.4. In addition, with application-specific keys, the security of the encrypted keys still relies on Pearl-TEE's ability to maintain the secrecy of the storage encryption keys, except there are now more keys to protect instead of one. As a result, using different storage encryption keys for each payment application only increases complexity without any apparent benefit. Finally, with an application-specific storage key, the number of keys to protect scales with the number of applications, while having a single storage key makes the number of keys application-independent. This design enables Pearl-TEE to provide a functionality-specific replacement for the complexity of a commodity OS FS, which may contain over 36K lines of code (ext4), with a simple decryption operation.

**M-2: Batch execution**. Commodity OSs use fairly complex interrupt-driven CPU multiplexing and scheduling to share execution resources among applications while isolating application execution state. A great deal of complexity associated with this is the result of having to ensure that all necessary parts of the application state are saved so that the application state can be resumed after being interrupted. However, we expect that payment applications will rarely run concurrently as users are likely to only make one payment at a time. In addition, we do not expect payment applications to run for long as they only need to construct a payment message and hand it to Pearl-TEE for further delivery to the server. As a result, Pearl-TEE uses batch execution for payment applications and assumes that they can be terminated without the need to resume them if they run for too long. This removes the need for the code to save and restore application execution state, as well as the need to implement a scheduler.

**M-3: Fairness timeouts**. To use batch execution means that Pearl-TEE cannot use interrupt-driven multiplexing to share execution time on the CPU. However, given that payment applications are not performance-critical, and should complete a transaction within a bounded time, Pearl-TEE implements a simple timeout, after which it will terminate a payment application, to ensure fairness. To measure the amount of time an application has executed, Pearl-TEE leverages the hardware timers available in the CPU.

### 4.2.4 System Overview and Operation

Pearl-TEE only protects and executes the trusted components of the payment application (we call it *TEE component* hereinafter) that is meant to execute inside the TEE. Payment applications may implement untrusted, application-specific functionality outside of the TEE component in a normal-world application component (we call it *normal-world component* hereinafter). We differentiate between these components as shown in Figure 4.2. An untrusted normal-world component interacts with the application requesting the payment, prompts the user to input payment details and sends these details to a TEE component. The TEE component verifies the user's credentials, confirms the payment details with the user, retrieves a user-specific signing key from Pearl-TEE and signs the payment details to make a payment message. This payment message is then sent back to the normal-world OS for transmission to the payment server via a kernel module in the normal-world OS. The normal-world component may be compromised by an adversary without a compromise of the payment process since all payment details are confirmed and acceptance is verified via the user providing credentials before the payment message is signed. The normal-world components are purely for ease of use.

Operation of payment applications using Pearl-TEE proceeds in three major phases: Device Trusted Boot (DTB), which happens every time the device boots; Initial Software Setup (ISS), which occurs the first time a payment application is installed; Device Payment Flow (DPF), which happens each time the user makes a payment; and TEE Component Update (TCU), which occurs every time the application needs to update the TEE component binary. We describe each of these in more detail.

#### Device Trusted Boot

Even for single-tenant payment systems, the device must be able to attest to the payment server that the payment application is running in TEE. To do this, existing TEE payment applications, such as Samsung Pay, employ a trusted boot sequence. In such a sequence, the TPM (or secure element) computes a hash of the first software element to boot, which is normally the bootloader. In the case of a TPM, the hash is stored in a Platform Configuration Register (PCR) on the TPM. The bootloader then computes a hash of the next binary to load and combines this hash (a process called "extending") with the current hash in the PCR. Each subsequent binary computes a hash of the next binary to load and extends the hash stored in the PCR until the boot process is complete. To attest the identity of the software that is running on the system and the sequence that it was loaded in, the software may request the TPM to produce a "quote" of the value in the PCR, which is simply a signature over the contents of the PCR. As a result, one can see that since the TPM is trusted and performs the first hash measurement, and each component adds the hash of the next one, any untrusted or malicious software component loaded during boot will necessarily have its hash included in the PCR by the previous component, and thus a remote party will be aware of it.

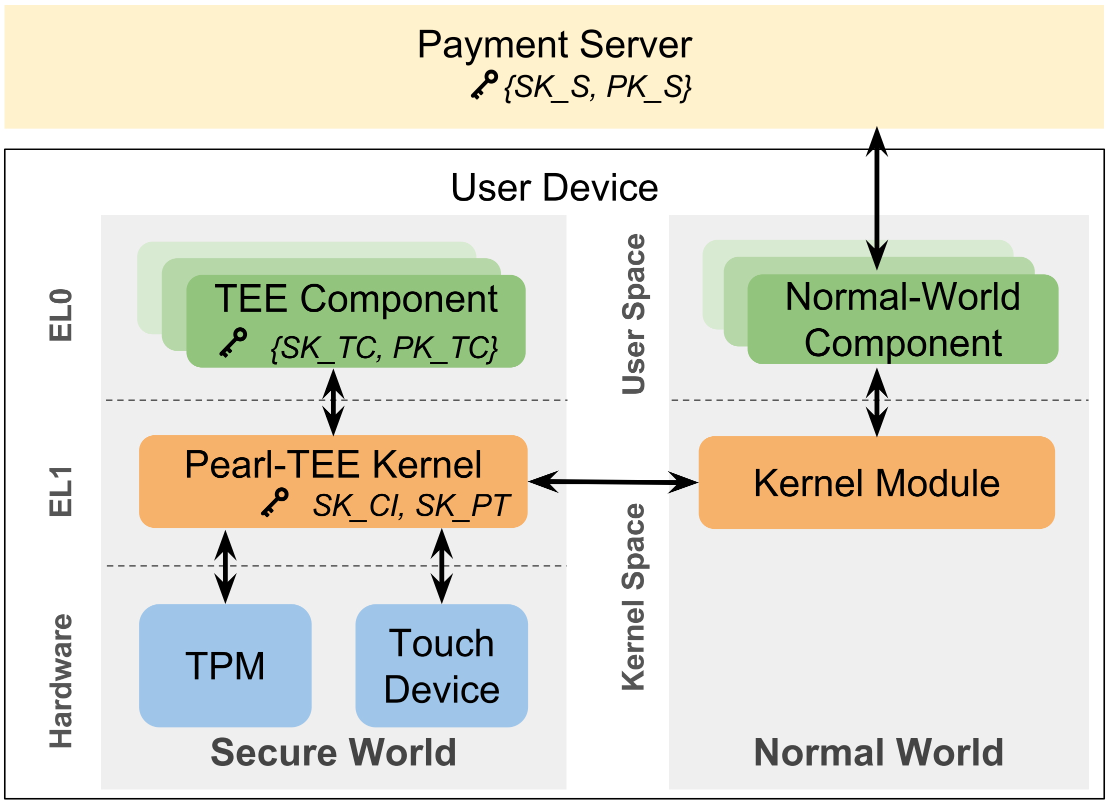


Figure 4.2: Mobile payment framework system design overview.

Since Pearl-TEE must maintain all guarantees of single-tenant systems, as well as provide guarantees required for secure multi-tenant systems, Pearl-TEE also employs trusted boot in the same way so that it may attest to a remote party that payment application TEE components are running in a trusted environment. In addition, since all trusted components must run before untrusted components in the trusted boot sequence, Pearl-TEE is loaded by the bootloader, which subsequently loads the normal-world OS. Finally, Pearl-TEE has a secret key `SK_PT` for encrypting data that needs to be stored persistently inside the normal-world OS. To store this key persistently across reboots, Pearl-TEE uses the TPM's sealing functionality, which makes the key only accessible to Pearl-TEE when booted in the correct sequence.

### Initial Software Setup

The first time a user uses a payment application she needs to perform Initial Software Setup (ISS). We assume each payment application has corresponding payment server and that both application and server are provided by a payment provider. Signed payment messages that result from well-formed and user-authenticated payment request from the normal-world component to the TEE component are then sent by the normal-world OS to the payment server, which then accepts and records the payment and transfers funds.

ISS binds a device with an instance of a payment application to a particular user and transmits this binding securely to the payment server for use in future payment transactions. This process begins the first time a payment application is executed on a device. First, the payment application requests user authentication by providing their account credentials, which include a secret password. Since the user is providing authentication credentials to a normal-world application, this phase requires that the normal-world is not compromised and that the user is using an authentic normal-world application. Next, the normal-world component will transfer execution to the TEE component in Pearl-TEE via the kernel module. This call specifies the location of the TEE component binary for Pearl-TEE to execute, as well as a certificate retrieved from the payment server so that the TEE component can establish a secure channel with the payment server. Upon execution, the TEE component verifies the authenticity of the payment server certificate. This can be done in a variety of standard ways—for example, it could verify that the certificate was signed by an appropriate CA or it could use a pinned certificate. The TEE component then generates a key pair {`SK_TC,PK_TC`}, retrieves the user's username and credentials via the trusted path and provides both to the Pearl-TEE.

Pearl-TEE generates an application-specific key `SK_CI` that combines its secret key `SK_PT` and hash `#H` of the payment application TEE component. Using an application-specific key that depends on the hash of the TEE component saves Pearl-TEE from having to explicitly associate encrypted application key-pairs with the identity (hash) of each application TEE component. If a faulty or tampered TEE component presents an encrypted key pair it does not own to Pearl-TEE for decryption, the key pair will simply not decrypt correctly and the faulty component learns nothing about the true TEE components `SK_TC`.

The TEE component then starts to register its public key, `PK_TC` with the payment server by generating an uncertified ISS message, *ISS_msg* = {*username, enc(credentials), PK_TC*}, which binds the public key of the TEE component to the user using the application. The TEE component then requests that Pearl-TEE certify it by creating an attestation that includes a measurement of the Pearl-TEE environment and the hash of the uncertified ISS message and TEE component binary. The final certified ISS message contains the uncertified ISS message and an attestation: {*ISS_msg, #H, Attest(Pearl-TEE, Hash(#H | ISS_msg))*}.

The certified ISS message must be transmitted to the payment server over a secure channel as it contains sensitive user credentials. Pearl-TEE establishes an SSL connection via the normal-world kernel module with the payment server, uses the server certificate to verify it is communicating with an authentic payment server and transmits the certified ISS message. The message certifies to the server that:

1. the ISS message originated from an instance of Pearl-TEE running in a correct TrustZone TEE,

2. the ISS message originated from a TEE component with binary hash `#H` and,

3. the TEE component wants to associate the public key `PK_TC` with the user `username`. The server decrypts the user credentials and verifies that they match that of the user associated with `username`. If they match, the server knows that a valid user initiated the ISS message as well. At this point, the server sends back confirmation that it has accepted the `PK_TC` of the TEE component.

From this point onward, the payment server uses verification of challenges with `PK_TC` in lieu of authenticating the user again. This is secure because the ISS process ensures that the matching signing key `SK_TC` is only available to a particular instance of a TEE component running in TEE and bound to a particular user. In addition, the `PK_TC` uniquely identifies each instance of the TEE component and allows for easy revocation of any instance. The TEE component must store its `SK_TC` so that it may use them on subsequent payment transactions. To do this, it submits them to Pearl-TEE, which encrypts them with the application-specific key `SK_CI` and gives the encrypted values to the TEE component, which stores them on the normal-world FS along with `PK_TC`. On subsequent payment transactions, the TEE application can retrieve the encrypted values and request Pearl-TEE to decrypt them. The values will only decrypt correctly if they are submitted by the correct TEE component because the hash of the TEE component forms part of the key that was used to encrypt the values.

The ISS phase only happens at payment application user setup for the first time and when the payment application needs an update, which we will discuss as part of the TEE Component Update (TCU) phase later in this section.

## Device Payment Flow

All payment transaction are abstracted into a payment message, that is sent from the TEE component to the payment server. A payment transaction is any function that requires secure communication between the device and the payment server and can include transferring funds, depositing new funds, confirming previous payments or changing account settings. To perform a payment transaction, Pearl-TEE and the payment application perform the Device Payment Flow (DPF). Figure 4.3 shows the four major steps in the DPF. Note that all network messages from the secure world to the payment server are sent via the kernel module in the normal-world OS as in Figure 4.2, which we omit here for convenience. In order, the four steps are:

**Collect transaction details.** User initiates the payment transaction ① and submits user transaction details, which specifies both the type of transaction and parameters specific to that transaction. For example, for a simple payment transaction, the user might specify a payee, an amount and an account to pay from, while for requesting proof of payment to prove to a third party that payment was made, the transaction details might include a transaction number, a date and a payee. After collecting this information from the user, the normal-world component establishes a connection with the payment server and the latter responds with a one-time session token $m$ ②, which the normal-world component delivers to Pearl-TEE ③. $m$ acts as a nonce to protect payment transactions against replay. The normal-world component then submits the transaction details, $m$, TEE component binary, the application's `PK_TC` and the encrypted `SK_TC` to Pearl-TEE.

**User confirmation.** Pearl-TEE computes the hash `#H` ④ of the TEE binary and uses this to compute the `SK_CI` for the TEE component. It then decrypts the encrypted `SK_TC` with the computed `SK_CI` and then transfers execution to the TEE component providing the `PK_TC`, the decrypted `SK_TC`, transaction details and $m$ ⑤. The TEE component then needs to confirm the transaction details with the user. The TEE component may perform some application-specific formatting on the transaction details and uses the trusted display to display them to the user along with a request that the user touch the touch device if they agree with the displayed details ⑥. If the user does not agree with the transaction, she will hit another button (i.e. the power button). We note that while the confirmation channel (i.e. the touch device) needs to be trusted so that a malicious application cannot forge a confirmation from the user, the abort channel need not be. Payment transactions already depend on an untrusted normal-world application to start them anyways, so giving a malicious application to abort the transaction does not weaken the security guarantees of Pearl-TEE. The user will still be informed by Pearl-TEE that the transaction has been aborted.

We also note that we do not need a trusted indicator that the display is being used by the TEE component as in other systems that rely on trusted indicators [164, 103]. This is because the TEE component will always display the transaction details and request user confirmation. While a malicious normal-world or TEE component could try to spoof this display, it cannot prevent the TEE component from displaying its confirmation request or the transaction will not be completed. Thus, so long as the user does not confirm any transaction they do not wish to execute, a malicious application can display arbitrary information but not be able to cause the legitimate TEE component to perform a transaction the user does not approve. Once the user approves by activating the touch device, which has a trusted path to the TEE ⑦, a signal is sent to Pearl-TEE.

**Certify payment message.** With user confirmation received, the TEE component then prepares to certify the transaction details and create a payment message to transmit to the payment server. The TEE component constructs a payment service-specific payment message for the requested transaction, including the nonce $m$ to guarantee freshness, and signs the entire transaction with its `SK_TC` and then returns the signed payment message to the normal-world component via Pearl-TEE kernel ⑨. Note that unlike in ISS, the TEE component need not create an SSL channel even if the contents of the payment message contain sensitive information. Instead it depends on the normal-world component to transmit the details to the payment server via SSL ⑩. This is because, if malicious, the normal world component will have learned all the payment details already anyways, as the only piece of information the normal-world doesn't have is the `SK_TC`, which is only used to sign the payment message but never transmitted out of the TEE component. One might be tempted to move the entire entry of the payment details by the user into the TEE component, but in practice, this does not work as often some of the payment details are specified by third party applications also running in the normal-world. For example, the user may use a third-party application that triggers a Paypal payment specifying the payee and amount when the user wishes to pay for some item from the third-party.

**Payment transaction confirmation.** The payment server receives the signed payment message and verifies its authenticity using the `PK_TC` in the message and by checking that $m$ has only been used once to prevent replay. Note that the `PK_TC` uniquely identifies the user and device making the payment transaction as setup during the ISS phase. If the `PK_TC` is not a valid `PK_TC` set with a successful ISS operation, or the signature or nonce do not verify correctly, the payment server returns an error. Otherwise, the payment server confirms success ⑪ and the transaction is completed. The normal-world component displays the confirmation to the user. One might wonder why the confirmation is not sent to the TEE component to display to the user so that a malicious normal-world component can't forge a confirmation message and make the user think the payment transaction completed when in fact it didn't. Unfortunately, to make it impossible to forge the confirmation a secure indicator would need to be used to indicate to the user that the confirmation was in fact coming from the TEE component, and the literature has not conclusively shown that users can be relied upon to pay attention to security indicators [45]. Rather than add complexity to both the device and TEE component, we feel that a simple solution might be to have the user check that important transactions are completed by checking with the payment service out of band at a later time (i.e. checking the payment server's website from their laptop). Alternatively, if the payment server supports payment confirmations, the user may request a payment confirmation that sends a specific payment server-generated confirmation number that can only be decrypted and displayed by the TEE component, making it hard for a malicious normal-world application to forge the confirmation.

To maintain a small TCB, Pearl-TEE does not have the ability to save TEE component state and restart them if they are interrupted. If the interrupt is vectored to the normal-world OS, the hardware transparently saves the state of the TEE into TrustZone storage allowing Pearl-TEE and the TEE component to be restarted after the normal-world OS is done handling the interrupt. However, if the interrupt is vectored to Pearl-TEE, the OS will destroy TEE component state in the process of handling the interrupt. In practice, this is not a problem since interrupts vectored to Pearl-TEE can only be generated by either the TEE component itself or devices connected to the TEE such as the touch device. Since Pearl-TEE only executes a single TEE component at a time, there can be no other TEE component that could generate an interrupt. To prevent a transaction being interrupted by a second errant touch, Pearl-TEE disables interrupts by the touch device and power button after user confirmation is received.

Finally, Pearl-TEE has to deal with malicious TEE components that never terminate and thus prevent any other TEE component from running. Before a touch is registered, a user can always terminate a TEE component by aborting the transaction, which sends an interrupt directly to Pearl-TEE. However, after the touch is registered, Pearl-TEE disables interrupts from external devices to the TEE. Thus, Pearl-TEE imposes a time limit $\Delta T$ after which it will forcibly terminate the TEE component after a touch on the touch device is registered.

### TEE Component Update

We assume that the payment server may want to occasionally update the software in the TEE component to provide new functionality or address software defects. Because the hash `#H` will change on such an update, this makes the `SK_TC` created during ISS inaccessible to the updated TEE component. To make the `SK_TC` available to the updated TEE component, the old TEE component can perform TEE Component Update (TCU). First, the old TEE component verifies the authenticity of the new TEE component. Pearl-TEE requires that the payment server sign the updated TEE component, which the old TEE component can verify using the same method it used to verify the server certificate during ISS. Once it verifies that the updated TEE component originated from the payment server, the old TEE component makes a call to Pearl-TEE to rekey the `SK_TC`, passing the encrypted `SK_TC` along with the hash `#H` of the new TEE component. Pearl-TEE decrypts the encrypted `SK_TC` using the `SK_CI` for the current TEE component and then re-encrypts it with the new `SK_CI` of the updated TEE component, generated with the passed in `#H` and Pearl-TEE's `SK_PT`. This new re-encrypted `SK_TC` can then be used by the updated TEE component.

The updated component should send a signed message to the server informing it of the update so the payment server can associate the new `#H` with the `PK_TC`. This way, the payment server always knows what version of the TEE component is associated with each `PK_TC`. This allows the payment

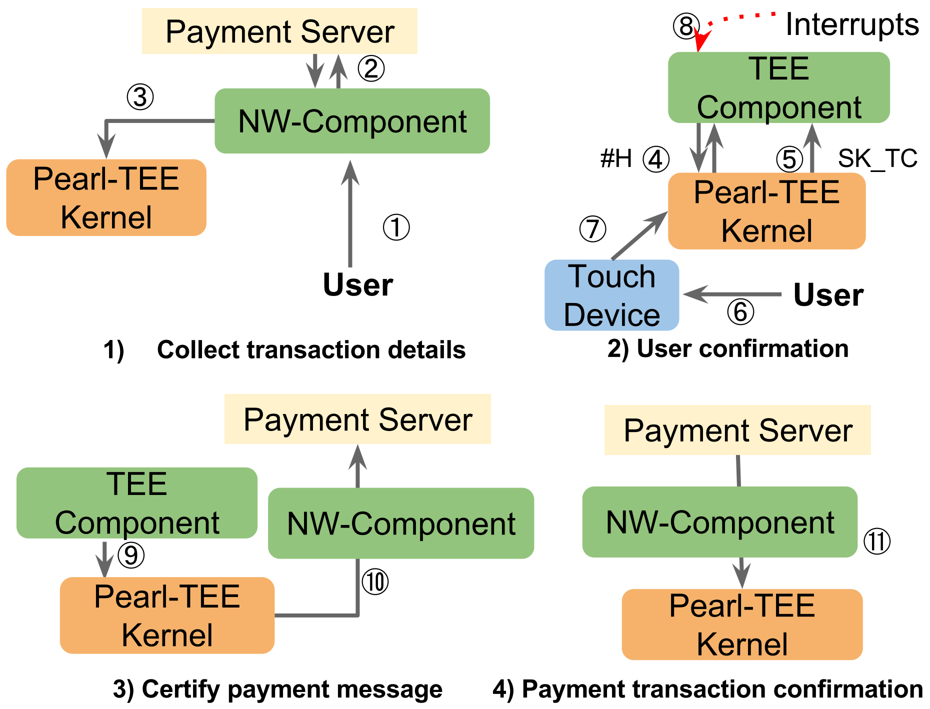


Figure 4.3: Interactions in the phase of Device Payment Flow

server to force an update of the TEE component when vulnerabilities are discovered.

On its own, this process is vulnerable to a rollback attack, where an adversary could force a victim to use an old, possibly vulnerable, version of the TEE component. There are two complementary ways to mitigate this. The first is that the certificate for each TEE component includes both the #H of the component and a version nuqmber and TEE components will only perform TCU if the updated component has a newer version number than the current component. The second is that the TEE component always informs the server of a TCU event, at which time the server can reject the update and disable transactions from that PK_TC until an update message to an appropriate #H is received. We note the latter method can let the payment server know that the TEE component has been updated so it will opt to ignore requests from an out-of-date one, moreover, it is useful in that it allows the payment server to notify a user if they are using an out-of-date TEE component and force an update as mentioned above. These two methods can be used individually or together as a defense-in-depth measure.

### 4.2.5 Security Analysis

We analyze the security properties of Pearl-TEE and discuss how the various mechanisms discussed enable Pearl-TEE to withstand various attacks.

**Resilience against malicious normal-world OS**. For simplicity, Pearl-TEE requires that TEE components use the normal-world OS for storage of persistent information, such as the payment message signing key SK_TC. However, this key is encrypted with a key known only to Pearl-TEE so a malicious normal-world OS cannot recover the plain text. The normal-world can corrupt the cipher text of the key, which will render the TEE component unable to sign any future payment messages.

However, this only impacts the availability of service, which Pearl-TEE cannot guarantee. Once created, `SK_TC` key does not change for the lifetime of the TEE component it is associated with. As a result, it is immune to replay attacks.

A normal-world OS can forge a payment request and submit it without the user's knowledge. However, the user confirmation phase of the DPF allows the TEE component to ensure that the user is aware of and approves the payment transaction before it signs it.

Because Pearl-TEE also depends on the normal-world OS to send network messages, the normal-world OS can interrupt DPF and ISS at any point. This can affect the availability of the payment application, and can also make it ambiguous whether a payment transaction completed or not. However, the ability to interrupt transactions does not give the normal-world OS the ability to forge transactions, and the user can always confirm whether a transaction completed or not through out-of-band means not involving the device. As a result, the ability to interrupt transactions does not impact any of the four guarantees described previously that Pearl-TEE provides.

**Resilience against malicious TEE component**. A malicious TEE component can try to impersonate another TEE component. However, this is impossible since Pearl-TEE uses the hash `#H` of the TEE component binary to identify it. Since the key, `SK_CI`, used to encrypt a TEE component's `SK_TC` depends on `#H`, a malicious TEE component cannot gain access to another TEE component's `SK_TC`. While a TEE component can trick a user into believing it is another TEE component, this does not help in that it still cannot forge a payment. We do not require the user to input any passwords during DPF so a malicious TEE component cannot phish any user credentials. The only time user credentials are used is during ISS, which only happens the first time a payment application is used. This presents a malicious TEE component an opportunity to steal the user's credentials if it is able to successfully impersonate both the normal-world component and TEE component of another payment application. Unfortunately, this is unavoidable and equivalent to the user installing a trojaned binary on their device. Thus, the user must be vigilant and ensure that they are using an authentic payment binary the first time they interact with that payment service.

Pearl-TEE currently does not require a trusted path for the user to enter their password and pass it to a TEE component. Even if such a trusted path existed, it would not remove the requirement that the TEE OS is uncompromised and the payment application is authentic during ISS. This is because a trusted path for user credentials is useless if the TEE component at the other end of the path is maliciously claiming to be another TEE component. The only way to ensure that user credentials are not phished is to ensure that the user is interacting with an authentic payment application and that the normal-world OS has not tampered with it.

Finally, a malicious TEE component could perform a fake TCU and request Pearl-TEE to re-encrypt another TEE component's encrypted `SK_TC`. However, the malicious TEE component can't use it's own `#H` binary as that binary and `#H` must be signed by the payment server. Thus, it has to use a legitimate `#H` and the `SK_TC` will be encrypted with that key, but if the malicious TEE component tries to decrypt it, our TEE OS will use the `#H` of the malicious component and invalidates the malicious attempt.

**Attack surface reduction**. By outsourcing all networking and persistent storage functionality to the normal-world OS, Pearl-TEE maintains a small TCB. By using batch scheduling, Pearl-TEE avoids having to save and restore interrupted TEE component state. In addition, Pearl-TEE does not have to maintain any descriptors that keep track of multiple execution states or a scheduling algorithm to arbitrate among them. This also simplifies the APIs that Pearl-TEE must maintain, allowing it to have a smaller attack surface.

## 4.3 Mint-TEE: Supporting Scalable Rollback Protection for TrustZone Applications

### 4.3.1 Design Objectives

In addition to our goals of liveness, freshness and scalability, we designed Mint-TEE with the goal of protecting TEE Apps from each other, and from a potentially malicious commodity OS. Additionally, Mint-TEE also isolates the commodity OS from potentially malicious TEE Apps by limiting a TEE App's access to the commodity OS's memory pages and allowing the commodity OS to interrupt TEE Apps to service interrupts. As a result, Mint-TEE has the following design goals:

- Isolation: Mint-TEE should isolate TEE Apps from each other, as well as from the commodity OS. Mint-TEE should also ensure that TEE Apps cannot access memory of the commodity OS.

- Liveness: We use liveness to denote the ability of TEE Apps and the commodity OS to eventually gain control of the processor in a timely manner when they need it. In particular, TEE Apps should not be able to starve the commodity OS or each other. However, since the execution of a TEE App is always initiated by an application component in the commodity OS, Mint-TEE cannot prevent a malicious commodity OS from disallowing a TEE App from running. Finally, TEE Apps should be able to recover their state in the event of an unexpected power failure or commodity OS crash.

- Freshness: TEE Apps should be able to ensure that data stored in the commodity OS is fresh and has not been rolled back by the commodity OS to a stale state. We elaborate further on the motivations for storing data in the commodity OS below.

- Scalability: The services and security guarantees provided by Mint-TEE should be scalable to an increasing number of TEE Apps executing in TrustZone.

TrustZone provides Mint-TEE access to two kinds of resources: secure memory and peripherals, which are accessible only to Mint-TEE, and shared resources, which are accessible to both Mint-TEE and the commodity OS. In general, board designers will provide only limited secure resources, as they are wasted if unused by the Mint-TEE. For example, TrustZone typically only provides access to a small amount of memory and persistent storage (10's to 100's of MBs) [46]. Because Mint-TEE is designed to run an arbitrary number of third-party TEE Apps, we expect these secure resources to come under pressure.

Mint-TEE alleviates this pressure by borrowing the resources of the commodity OS. However, as mentioned in in previous section, this exposes TEE Apps to rollback attacks by a malicious commodity OS. To address this, Mint-TEE allows TEE Apps to create an arbitrary number of virtual counters to protect the freshness of state stored in the commodity OS, and a TEE App State Counter for each TEE App, to protect its execution state when stored in the commodity OS.

### 4.3.2 System Overview

We base our design of Mint-TEE on Pearl-TEE [76], an open-source TEE OS, based on OP-TEE [105], that is designed to isolate TEE Apps from each other and the commodity OS. Pearl-TEE provides several features which serve as a good starting point for Mint-TEE. Pearl-TEE uses privilege levels to isolate itself and TEE Apps from each other. Specifically, Pearl-TEE runs at secure exception level 1 (SEL1) and standard TEE Apps run at secure exception level 0 (SEL0). With hardware MMU support, this prevents Mint-TEE from being tampered with by malicious TEE Apps. Pearl-TEE does not allow TEE Apps to be co-resident in memory and only one TEE App may execute at a time, thus isolating TEE Apps from each other. Like Pearl-TEE, Mint-TEE requires all TEE Apps to be signed using keys certified by a known certificate authority and checks the binary hash of a TEE App when it is loaded into memory. This mechanism is similar to other TEE OSes that support arbitrary TEE Apps [171].

To enable communication between TEE Apps and the commodity OS, Mint-TEE adds a request queue and a response queue to Pearl-TEE, as well as a shared memory region between Mint-TEE and the commodity OS for passing large data values. Requests can be made by both TEE Apps to the commodity OS and commodity OS application components, as well as by Mint-TEE itself to the commodity OS. To use the queue, requests are placed onto the request queue, which may include references to locations in the shared memory region. Mint-TEE, then transfers control to the commodity OS by issuing a non-secure interrupt by sending an *IRQ*. The interrupt is handled by the secure monitor, which saves the TEE OS context, restores the commodity OS context and indicates an *IRQ* is to be delivered with the state vector `VBAR`. The commodity OS receives the requests and either fulfills them itself, or forwards them to the appropriate commodity OS application. Responses are placed onto the response queue and can be parsed and consumed by Mint-TEE or the appropriate TEE App the next time the commodity OS invokes Mint-TEE. The details of the procedure of how TEEs and commodity OSes communicate can be found in the ARM GICv3 document [6]. Requests and responses need not be matched. As an example, TEE Apps are first invoked by commodity OS applications by enqueuing an appropriate message onto the response queue.

By enabling this communication, Mint-TEE provides a convenient way for TEE Apps to access commodity OS resources. For example, a TEE App can request its commodity OS application component to save some encrypted data to a file. Similarly, if one TEE App is interrupted and another TEE App needs to execute, Mint-TEE provides a limited form of concurrency that Pearl-TEE does not support by saving the encrypted and signed state of the interrupted TEE App to the shared memory region, and then using the secure memory to execute the new TEE App. This effectively allows Mint-TEE to use the memory resources of the commodity OS as swap space for TEE Apps.

Both of these use cases are vulnerable to rollback attacks, where a malicious commodity OS can restore a stale file or suspended memory state of a TEE App. However, persistent TEE App data stored to a file and suspended TEE App state have different freshness requirements, which motivates the design of two different virtual counter systems: one for persistent storage and one for TEE App state.

### 4.3.3 Virtual Counters for persistent storage

Mint-TEE exports a virtual counter interface to TEE Apps that provides the following functions:

- **Create:** Creates a new virtual counter and returns a unique ID for the counter.
- **Read/Increment:** TEE Apps can read or increment a counter by specifying its ID.
- **Destroy:** TEE Apps can de-allocate a counter rendering it inaccessible in the future.

While we expect TEE Apps to use virtual counters to protect persistent data stored in files in the commodity OS from rollback, these virtual counters can be used the same way any hardware monotonic counter is used. For example, they can also be used to keep track of the number of times certain events have occurred, such as invocations of the TEE App or of certain TEE App functionality.

Mint-TEE and the commodity OS cooperatively implement virtual monotonic counters for TEE Apps using an interface which describes the separation of responsibilities and trust relationship between Mint-TEE and the commodity OS. Since only the commodity OS has access to persistent storage, it implements a `Store_VCs` function that can be invoked by Mint-TEE using the request queue described in Section 4.3.2 to store its virtual counters to persistent storage. Mint-TEE implements the `Seal_VCs` function, which creates and stores a token that protects the freshness of all virtual counters by linking them to a single hardware counter increment. Therefore, while Mint-TEE guarantees freshness of virtual counters, the storage requirements are completely satisfied by the resources of the commodity OS instead of Mint-TEE.

Mint-TEE uses a hash tree mapped into a region of memory that is shared between the commodity OS and Mint-TEE. Each leaf node in the hash tree represents a virtual counter, and the hash tree is ordered so that each virtual counter can have a unique index within the hash tree. The structure of the hash tree can be seen in Figure. 4.4. TEE Apps can make arbitrary read and increment operations on the virtual counters by invoking the corresponding Mint-TEE APIs. These cause Mint-TEE to read or increment the appropriate nodes in the hash tree. Reads on virtual counters in the hash tree are accompanied by a full hash checking of all nodes between the virtual counter and the root of the hash tree. The root of the hash tree is protected from rollback by a signature that includes a counter value that should match the state of a hardware monotonic counter $MC_p$. As part of Mint-TEE's increment aggregation, virtual counter increments occur on the hash tree and are not persisted by the commodity OS until Mint-TEE deems it necessary, as we describe below. When Mint-TEE does need to persist the virtual counters, it calls the `Seal_VCs` to persist them to disk and protect the committed virtual counters from being rolled back.

The `Seal_VCs` computes a MAC of the current value of the hash tree root and the value of $MC_p$ incremented by one and stores this along with the hash tree in the shared memory region. Mint-TEE then invokes the `Store_VCs` in the commodity OS, which stores the current contents of the shared memory (sealed hash tree root, hash tree, counter values, and the VCM) to persistent storage. If successful, the commodity OS sets a flag in the shared memory region. On its next invocation, Mint-TEE verifies the state of this flag and, if successful, increments $MC_p$ to match the counter value that was sealed to the root of the hash tree.

Because TEE Apps are mutually distrustful, Mint-TEE implements access control so that each virtual counter can only be read or incremented by the TEE App that created it. Currently, Mint-

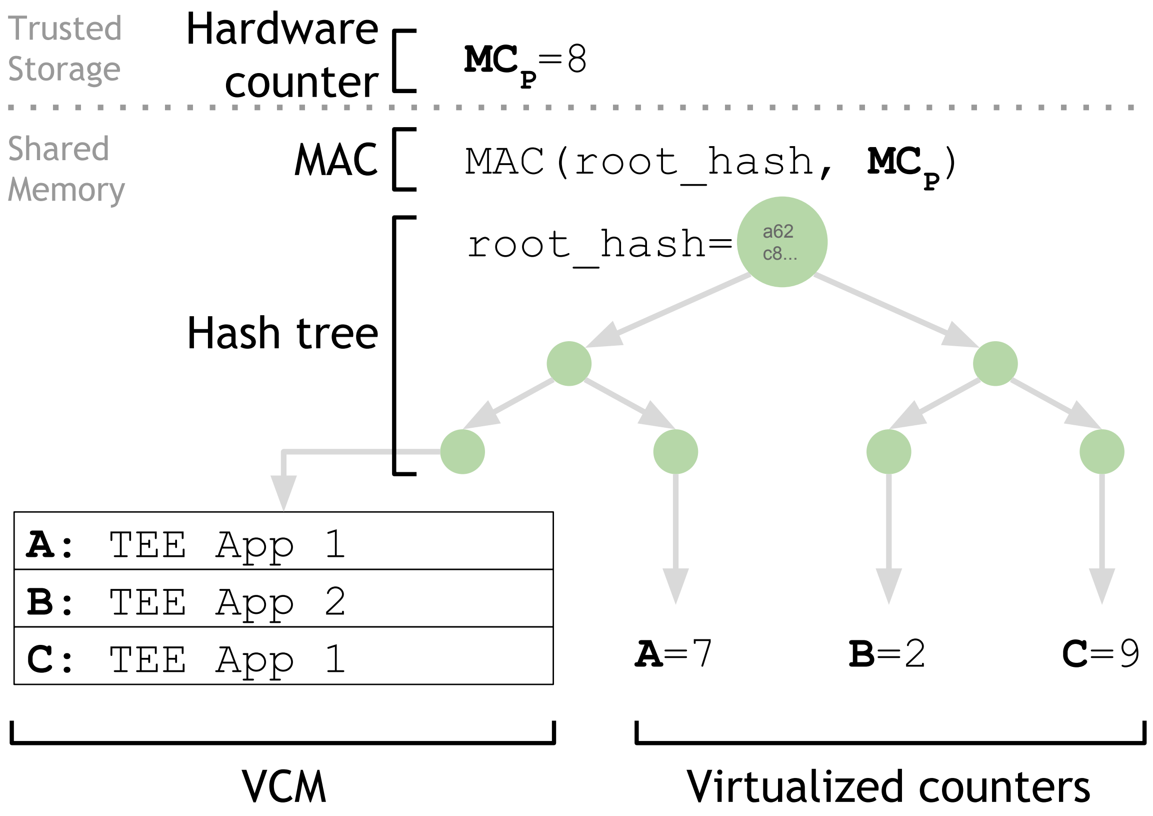


Figure 4.4: Illustration of the Mint-TEE hash tree used for storing virtual counters, as it appears in shared memory. It relies on a single hardware counter $MC_p$.

TEE does not permit virtual counters to be shared across multiple TEE Apps, though this could be implemented in the future. The access control list for each counter is stored in a *Virtual Counter Map (VCM)*. The VCM records, for each virtual counter, a mapping between its index and a tuple consisting of the identity of the TEE App (by its code hash) and the counter ID assigned to it when it was created. A counter index that does not appear in the VCM is currently unused (it has neither been allocated or is a virtual counter that has been destroyed). The VCM is stored in the same shared memory region as the hash tree and the first leaf in the hash tree is used to store a hash of the VCM to protect it from tampering or replay.

The VCM is updated whenever a virtual counter is created or destroyed. It is also updated under two other circumstances. First, the hash tree is initialized to contain some number of leaves (currently 256), but if the number of created virtual counters exceeds that number, the hash tree can be increased in depth to increase the number of leaves. In such an operation, the index of a virtual counter need not change as the new virtual counters are simply assigned indices higher than the index of the largest virtual counter before the depth increase. However, if subsequently, enough counters are destroyed so that it is possible to reduce the depth of the tree, Mint-TEE may choose to remove leaves from the tree. In this case, nodes are garbage-collected and the mapping between index and the $\{TEEApp, ID\}$ may change. Second, if the code of a TEE App is updated but needs to retain access to to the old code's virtual counters, Mint-TEE may update the TEE App ID for all of the App's virtual counters. Currently, Mint-TEE permits this if the code of the old TEE App and updated TEE app are signed by the same key. We expect updates to the VCM to be infrequent.

**Counter increment aggregation**: Hardware monotonic counters have a small, finite capacity that once exceeded, renders them useless. As a result, Mint-TEE's virtual counter design also attempts to reduce the number of `Store_VCs` and `Seal_VCs` operations, so that the number of increment operations on the hardware counter, $MC_p$, is also reduced. To do this, Mint-TEE uses a technique called *counter increment aggregation*, which opportunistically aggregates several virtual counter increments into a single $MC_p$ increment when `Seal_VCs` is invoked.

There are two opportunities for counter aggregation: (1) One TEE App performs consecutive increments to one or more virtual counters, which we call *intra-app counter increment aggregation*, and (2) several TEE Apps increment their virtual counters, which we refer to as *inter-app counter increment aggregation*. If these increments are all done without control being transferred to the commodity OS between them, then a single increment of $MC_p$ would be sufficient to protect all the virtual counters from rollback. This enables several virtual counter increments to be reduced to a single increment of the hardware counter.

Figure 4.5 shows how Mint-TEE executes `Store_VCs` whenever it is about to transfer control to the commodity OS, which can happen when a TEE App terminates and there are no other TEE Apps waiting to run (as in the figure), or when Mint-TEE receives an interrupt that must be serviced by an interrupt handler in the commodity OS. In both cases, Mint-TEE aggregates virtual counter increment requests after every write to persistent storage. Only after a switch to the commodity OS is initiated, Mint-TEE persists the virtual counter increments by calling `Seal_VCs()`, which invokes `Store_VCs` in the commodity OS to persist the virtual counters, and then increases the hardware counter $MC_p$ upon a successful return from the commodity OS. In figure 4.5, the applications write to storage directly, by asking Mint-TEE to forward their request. Mint-TEE aggregates the counter increments of TEE App 1 and TEE App 2 and only stores and seals the incremented counter values

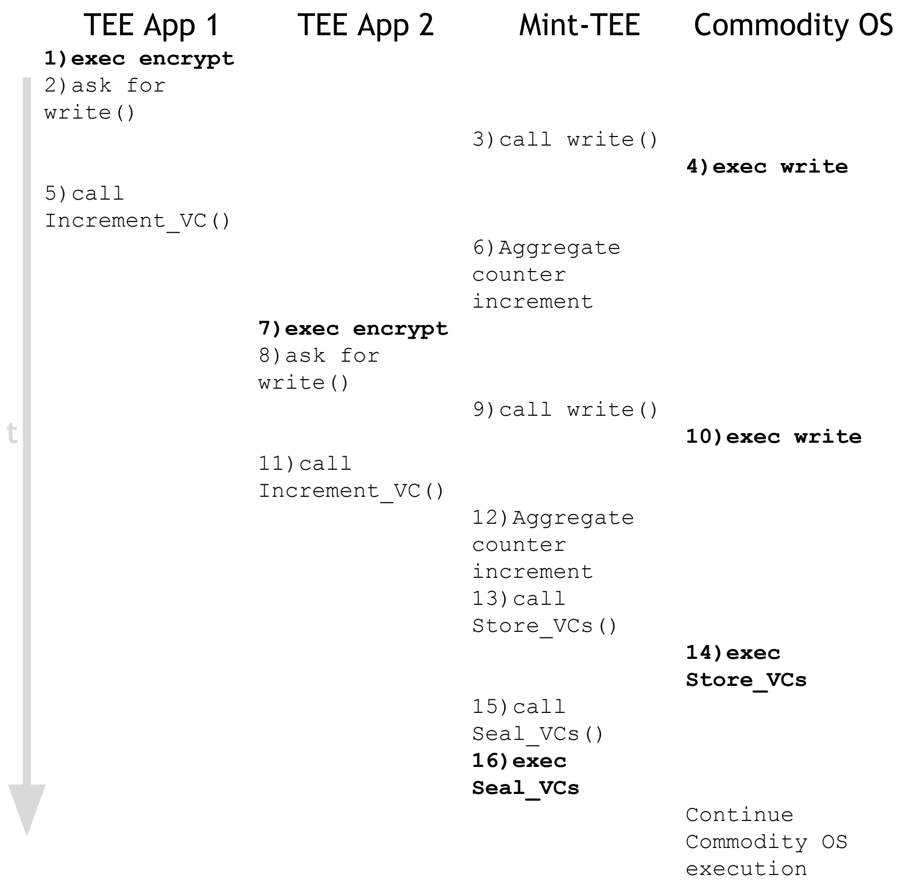


Figure 4.5: Steps taken by Mint-TEE when two file writes by different TEE Apps are performed.

once both applications finish executing.

**Hardware counter aggregation**: The above virtual counter design can be extended to use multiple hardware counters in a straightforward way. If there are $K$ hardware counters $MC_P[0], MC_P[1] ... MC_P[K]$, Mint-TEE establishes an ordering of the counters, and then increments one until saturation before moving onto the next one in increasing order (i.e. starting with counter 0, then 1 and so forth). Mint-TEE also adds to the sealed hash tree root an index $i$, which indicates which $MC_P[i]$ is currently being used. In this way, Mint-TEE aggregates multiple hardware counters into a single hardware counter and allows multiple virtual counters to share the single hardware counter resource.

**Recovery from system failures**: Because the signing of the hash tree root is not atomic with the increment of the hardware monotonic counter, a system failure due to loss of power or an OS crash may cause the hash tree and signed root to be written to disk without the hardware counter being incremented. In writing the hash tree root to disk before incrementing the hardware monotonic counter, Mint-TEE guarantees that the system can recover after the failure by having the commodity OS keep a copy of the previous state of the hash tree and restoring back to this state if Mint-TEE finds the hardware counter is one behind the current state of the hash tree.

The price of this recoverability is that it also allows a malicious commodity OS to rollback one set of updates by mimicking a power failure—an attack that Mint-TEE cannot distinguish from a real power failure. Furthermore, the commodity OS can use this ability to mount a *dictionary attack* [162], where it provides different inputs each time the victim state gets rolled back, thus learning the output of a TEE App for an arbitrary number of inputs, where the TEE App may have wanted to limit the number of responses it gives to the commodity OS using the virtual counter. A typical example of such an attack might be on a TEE App that wants to limit the number of authentication attempts. Normally the TEE App might increment a virtual counter after each attempt and lock the account after some number of attempts. However, by rolling back the state just before each increment of the corresponding hardware counter, a malicious commodity OS can also rollback the virtual counter and gain the ability to make an unlimited number of authentication attempts.

Vulnerability to such dictionary attacks is a well-known limitation of hardware counters, which can be addressed by incrementing the hardware counter before processing inputs, making operations deterministic so that they can only produce the same result if replayed by the commodity OS [133], or incrementing the hardware counter twice [162]. Unfortunately, all of these solutions have their drawbacks. Incrementing the hardware counter first means that recovery may be impossible after a power failure as the hash tree corresponding to the hardware counter may not have been committed to disk yet and thus permanently lost. Incrementing the counter twice means that the hardware counters will wear out twice as fast.

To solve this conundrum, we observe that not all applications are vulnerable to dictionary attacks. In particular, a dictionary attack is of no consequence if the adversary gains nothing by observing different possible states of the application. For example, unlike the case of being able to repeatedly make authentication attempts, a user may be annoyed if a malicious commodity OS repeatedly rolls back her secure chat logs, but is probably not hurt so long as the confidentiality of those logs is protected. Similarly, the ability to mount a dictionary attack on the Mint-TEE hash tree is in general of limited value to an adversary. Rather, the adversary wants to mount a dictionary attack on a specific TEE App that uses virtual counters to limit sensitive operations such as authentication attempts. Thus, we enable TEE Apps to protect against rollback by allowing them to use the double increment defense proposed by Ariadne [162] if they require protection from dictionary attacks. If control is transferred to the commodity OS in between the double virtual counter increment, then both increments will be translated to hardware counter increments. However, if there is no interrupt between the two increments, then the commodity OS would not be able to rollback the state and both of the virtual counter increments would be aggregated into the next hardware counter increment, thus reducing the number of hardware counter increments.

### 4.3.4 TEE App State Counters

As discussed earlier, Mint-TEE saves the state of suspended TEE Apps into a shared memory region backed by commodity OS memory. Even though Mint-TEE encrypts and signs the state, a malicious commodity OS can reset a TEE App back to an earlier state by rolling back the suspended state to an earlier state. Coupled with the ability of the commodity OS to schedule interrupts that would return control back to it (i.e. a commodity OS has the capability of issuing an `FIQ` or `IRQ` interrupt arbitrarily), this allows a malicious commodity OS to cause an executing TEE App to be suspended, and then start another TEE App to cause the executing TEE App to have its state suspended to memory, allowing the commodity OS to then replace the suspended state with a previous one. This could allow a malicious commodity OS to rollback critical states or operations of TEE Apps, such as view counts in a DRM application, or transactions in a mobile payment application. Moreover, if the malicious commodity OS could rollback the runtime state of an arbitrary TEE App, it could possibly hijack the TEE App and force it to repeat certain operations to mount a DDoS attack on some other victim.

To prevent a malicious commodity OS from rolling back the state of suspended TEE Apps, Mint-TEE implements a TEE App state counter (TASC) to protect app states. The requirements of the TEE App state counter are quite different from that of virtual counters for TEE App persistent storage. First, the TASC does not need to be live after a power failure or commodity OS crash as the memory state of applications is assumed to be invalid across reboots. Second, the TASC need not be explicitly read or incremented by TEE Apps themselves, as the suspending and restoring of interrupted TEE App states should be transparent to TEE Apps. Mint-TEE only allows one instance of each TEE App to exist at any given time. As a result, Mint-TEE needs to support at most one TASC per TEE App.

Similar to virtual counters for persistent state, Mint-TEE must be able to handle an arbitrary number of TASCs as there could be an arbitrary number of TEE Apps installed. Mint-TEE implements TASCs using a hash tree implementation similar to that of the persistent virtual counters, where each leaf in the tree corresponds to a TASC for a particular TEE App. Similarly, Mint-TEE uses a VCM, also protected by the hash tree to map each leaf in the hash tree to a TASC. As in the persistent state case, an increment of a leaf counter causes updates all the way to the root of the hash tree and any leaf read causes verifications all the way to the root of the hash tree.

However, the implementation of the TASC hash tree also differs significantly from the hash tree to implement persistent virtual counters. Because the hash tree need not be recoverable across reboots, the root of the tree can be saved in TrustZone secure memory, which is protected from tampering by the commodity OS but as with commodity OS memory, also does not survive a reboot. This means that all TASCs will be reset to zero on each reboot. On its own, this would prevent rollback of states within a single boot, but would allow a malicious commodity OS to save a state from one boot, and then rollback to it on another boot. To prevent this, we make each TASC unique to each boot by using a single hardware monotonic counter. Thus, in reality, each TASC is really a concatenation of leaf values of the TASC hash tree and the value of the TASC hardware monotonic counter, which is incremented by Mint-TEE on each boot.

Mint-TEE uses TASCs to protect suspended TEE App state from rollback by signing each suspended state along with the current TASC value, thus sealing it as the most current state, and invalidating all other TEE App states, including those from previous reboots, that will have been sealed to other TASC values. As a result, a malicious commodity OS will not be able to have Mint-TEE restore any suspended TEE App state except the most recent one. In addition, Mint-TEE need not guarantee any ability to recover from a power failure or commodity OS crash since we do not expect the suspended execution state of TEE Apps to be preserved across such events (similar to commodity OS application semantics).

The design of TASCs is critical to enabling concurrency for TEE Apps, yet has low resource requirements in terms of hardware monotonic counters. Only a single such counter is needed. In addition, only a single increment per boot is needed, meaning that the counter is likely to last for many years, even with fairly frequent reboots, which would be sufficient for most devices we envision

Mint-TEE to run on, such as smartphones.

## 4.4 Implementations

Both of the Pearl-TEE and Mint-TEE prototype run on a LeMaker HiKey development board, which has a HiSilicon Kirin 620 SoC with an 8-core ARM Cortex-A53 CPU at 1.2GHz and 2GB RAM. We use Android AOSP 7.1 for the normal-world OS. We describe the details of the Pearl-TEE and Mint-TEE prototypes and how the example TEE applications are implemented in this section.

### 4.4.1 Pearl-TEE Prototype and Integration with Payment Services

Our implementation of Pearl-TEE is based on OP-TEE 2.0 [105], an open-source TEE that implements the GlobalPlatform TEE Internal API Specification v1.0. For cryptographic functions, we use the functions in the ARM Trusted Firmware v1.2 [7] library. TLS termination for network communications is implemented using the OpenSSL library v1.1.1 [129]. As required by security requirement **S-3**, Pearl-TEE kernel executes at exception level EL1, while TEE components run at EL0, allowing Pearl-TEE to protect itself from a malicious TEE component.

The GlobalPlatform APIs already provide a way for normal-world components to load and execute TEE components and invoke API functions in TEE components. As a result, our Pearl-TEE prototype needs only to implement the following APIs:

`ISS_Setup`: Used during ISS. Once the TEE component has generated it's keypair `SK_TC,PK_TC`, it passes them along with the user credentials to Pearl-TEE using this API call. Pearl-TEE then produces a certified ISS message with an attestation and also computes `SK_CI` for the TEE component and encrypts `SK_TC` with it.

`Decrypt_Key`: Used during DPF and TCU when the TEE component requests Pearl-TEE to decrypt its `SK_TC`.

`Display`: Used by the TEE component to display transaction details to the user during the transaction confirmation stage of DPF.

`Touch`: Used by the TEE component to verify user intent during transaction confirmation.

`NKM_Syscall`: Used by the TEE component to make system calls to the normal-world OS. It uses this API to send and receive network messages and to save encrypted keys to the normal-world file system.

To fully implement `NKM_Syscall`, Pearl-TEE proxies system calls to the normal-world OS in a way similar to Proxos [118]. However, while Proxos does it across a hypervisor, Pearl-TEE proxies system calls between the secure and normal-worlds of the ARM architecture. The Normal-World Kernel Module (NKM) receives `NKM_Syscall` requests from Pearl-TEE and sends them up to the Normal-World User Module (NUM), which executes them on behalf of the TEE component. The NUM runs as a dedicated, unprivileged user in the normal-world OS so that malicious TEE components cannot use it to escalate privileges in the normal-world OS. This design simplifies Pearl-TEE as it can proxy most complex functionality to the normal-world OS, while at the same time providing intuitive APIs for sending and receiving network message and storing persistent data, to the TEE component.

The Hikey board we use to prototype Pearl-TEE does not have a TPM or a touch device. Therefor we emulate these using software modules in Pearl-TEE. We emulate a TPM using a method similar to fTPM [143] with functionality based on a specification for ARM TPM modules [157]. We emulate a touch device similar to that in [164], which emulates a touch signal to Pearl-TEE.

To demonstrate Pearl-TEE, we integrate our prototype with the PayPal, Alipay and BrainTree mobile payment services. We select these three payment services because they are in common use, processing a total of over 1 trillion dollars in payments annually, and do not have implementations of payment applications that use TrustZone.

Pearl-TEE signs all messages with `SK_TC`, which must be set up with the payment server during ISS. Since we do not have control over the payment servers of these three payment providers, we implement a proxy server in front of their servers that will handle Pearl-TEE specific messages and uses the SDK's of the respective payment services to translate the Pearl-TEE messages into the current native payment messages of each provider. Because the proxy server sees the native SDK API calls to the payment server, it must be trusted. If Pearl-TEE is eventually deployed, any of these payment providers could support Pearl-TEE by simply running our proxy server themselves in front of their legacy payment servers. Our proxy server implementation is written in a combination of Python and Javascript.

To make payments to each of the services, we also need to write TEE components and normal-world components for each service. While each payment service offers several different types of payment transactions, our TEE components only support a basic payment transaction, as this is a common transaction that all services provide, making it easy for comparison in our evaluation. Even though payment transactions are all fairly similar at a high-level, application-level differences resulted in differences in the different TEE components for each payment service. For example, for AliPay, the TEE component needs to fill in the payment amount, transaction number, payment subject (a title of the transaction) and a notification URL to which the payment server should send the payment confirmation message. In comparison, for PayPal, the TEE component needs to fill in the type of transaction, account number of the user, payment amount and expiration time of the payment.

For future work, we intend to implement other types of transactions and other payment services. We explain two interesting examples of such transactions:

**Displaying balance confirmation**: In the Wechat Wallet service [166], after a payment transaction, the payment server confirms the payment together with a value $b$ indicating the current balance amount of a user wallet. In this case, the value $b$ should ideally be encrypted by the TEE component's public key `PK_TC` so that only the corresponding TEE component can decrypt it. However, the DPF process currently does not have a confirmation phase. Thus, we plan to implement this by having the normal-world component initiate a second "dummy" DPF that passes the encrypted $b$ to the TEE component, which can then display the confirmation to the user. When the user touches the touch device to continue execution, the second DPF terminates without sending any messages. The sole purpose of the second DPF is to display the confirmation balance $b$ to the user.

**Proof of payment**: Payment applications (e.g., AliPay) may request the payment server to provide a proof of payment, in a hashed string $p$ which is signed by the payment server, so that the user can later use it to prove to a third-party merchant that a payment/deposit is made. To request the proof, the normal-world component must collect details such as the payee, date and amount of the transaction and send these to the payment server.

### 4.4.2 Mint-TEE Prototype and System APIs

Our implementation of Mint-TEE is based on source code from Pearl-TEE [76] and it is compatible with GlobalPlatform TEE Internal API Specification v1.0. We upgrade the base system to OP-TEE v3.0 and port the changes made by Pearl-TEE. For cryptographic functions, we use the functions in the ARM Trusted Firmware v1.2 [7] library. The data structure of the hash tree that we use is implemented based on the Merkle tree library from the IAIK Secure Block Device Library [71]. TLS termination for network communications is implemented using OpenSSL library v1.1.1 [129].

The Mint-TEE prototype provides APIs based on the framework guided by the GlobalPlatform APIs. Mint-TEE does, however, contain extra functionality unique to Mint-TEE. The APIs pertaining to TEE App storage are summarized in Table 4.2. To safeguard TEE apps' persistent storage from rollback attacks, Mint-TEE provides an interface for TEE Apps to create, read, increment, and destroy virtual counters. The implementation of these interfaces follows the design outlined in section 4.3.3. Additionally, when Mint-TEE swaps TEE App in-memory state to ephemeral storage, it encrypts the state and increments a TASC. Similarly, the implementation follows from section 4.3.4.

To accommodate the growing number of persistent storage and accommodate multiple TEE Apps in TrustZone, Mint-TEE must support a dynamic number of virtual counters. Mint-TEE expands the hash tree once all existing counters become used up. To do so, Mint-TEE creates a new root node that points to the old root node as its left child and a new empty hash tree as its right child. This operation doubles the capacity of the hash tree, since the new empty hash tree on the right has the same size as the old hash tree on the left. The virtual counter map (VCM) is updated to account for the newly created virtual counters. The new root node is sealed by the Mint-TEE, while the old root node simply becomes yet another intermediate node of the hash tree. If the number of virtual counters in use shrinks dramatically, the hash tree is reduced in a similar manner, to save space.

The HiKey 960 development board we use for the Mint-TEE prototype does not have any hardware Secure Element (SE), monotonic counters or a touch screen. We therefore emulate these using software modules in Mint-TEE. We emulate an SE according to existing documents [112] that analyzes SE, and we emulate a touch screen device similar to that in TrustOTP [164] by emulating touch signals. For monotonic counter emulation, we take reference specifications from Microchip ATECC508A [117] and combine them with the SE module.

### 4.4.3 TEE Applications

We have ported two types of TEE Apps used in Pearl-TEE [76] to demonstrate the scalability of Mint-TEE's rollback protection: a payment app and a chat app, since both rely on persistent states. We believe these are representative TEE applications as users would likely be concerned about the security of their mobile payment transactions or the security of their chat messages. All apps contain both a commodity OS component and a TEE App that work together by communicating over the Mint-TEE message queue. For the protection of the persistent storage, we have modified them to use the APIs for accessing persistent virtual counters with the help of Mint-TEE. We briefly describe

Table 4.2: List of APIs Implemented for Secure Storage

| API Name | Caller | Description |
|---|---|---|
| Create_VC() | App | Create a new virtual counter and updates the record in VCM. |
| Read_VC() | App | Read the current value from a virtual counter that belongs to the requesting TEE App. |
| Increment_VC() | App | Increment the current value of a virtual counter that belongs to the requesting TEE App. |
| Destroy_VC() | App/OS | Destroy a virtual counter when it is no longer used, e.g., owner TEE App is uninstalled. |
| Store_VCs() | OS | Store virtual counters to persistent storage. |
| Seal_VCs() | OS | Increment $MC_p$ and bind with hash of the hash tree root. |
| Tree_Expand() | OS | Add one more level to the depths of the current hash tree and update VCM. |
| Tree_Reduce() | OS | Reduce one level of the depths of the current hash tree and update VCM. |
| TA_Save() | OS | Save a suspended TEE App state with encryption and bind with a TASC. |
| TA_Restore() | OS | Restore a suspended TEE App state after checking its freshness and integrity. |

how the new application flows work with Mint-TEE.

### TEE Payment Applications

Our implementation of the payment applications is also partially based on the work of the Pearl-TEE. In the base version, though not described in details on the application workflow, they achieve protection of confidentiality and integrity when communicating with commodity OS and the remote payment servers. The ported versions of the TEE payment Apps, compared to the previous versions, guarantee persistent storage freshness and liveness protection by making use of Mint-TEE virtual counters. In addition, Mint-TEE can provide a scalable number of virtual counters to different TEE Apps for satisfying their needs for storing multiple persistent files, meaning that the number of payment apps can continue increasing on a user's device. The scalability also applies to the number of TEE Apps the full system can support, as discussed earlier, suspended TEE App states are also under the protection of Mint-TEE.

To demonstrate a complete and understandable workflow of the TEE Apps, we describe the full payment framework, including the work done in the previous version and the new parts we added. The basic functionality of communication and guarantee of confidentiality and integrity was included in the original TEE Apps from Pearl-TEE, while the persistent storage protection with virtual counters usage are our newly added features to the original TEE Apps.

We use the SDKs of PayPal, AliPay and BrainTree mobile payment services to develop the payment TEE Apps in our prototype Mint-TEE system. To prove the authenticity of payment requests, our TEE applications send an attestation that includes the hash of the binary and payment request, proving that the request content was generated by a legitimate TEE App. Before accepting the payment request, the payment server should verify the attestation. Because the mobile payment servers are controlled by the payment service provider, and out of our control, we use a proxy server in between the mobile device and payment server to verify and forward the payment requests. Given that the proxy server sees the native SDK API calls to the payment server, it must be trusted. The proxy server implementation is written in a combination of Python and Javascript, depending on the example SDKs provided for the specific payment application. Each application consist of a portion implemented to run on the commodity OS (commodity OS application) and a portion in the TEE OS (TEE App). While each payment service offers several different types of payment transactions, our TEE application currently only supports a basic payment transaction, as this is a common transaction that all services provide.

Our mobile payment TEE Apps require persistent storage to store two types of files: (1) The transaction record, which is updated by logging transaction details such as the payee and payment account after each payment transaction, and (2) The key file, which stores the private key used by the TEE App to authenticate itself to a remote server. The key is a long-term secret that is only rotated periodically, but we can adjust its life span for the purposes of evaluation evaluation. Both of the above two files are private assets of the TEE App. They are only allowed to be accessed by the payment TEE App or Mint-TEE. To maintain confidentiality and integrity, these files need to be properly encrypted and signed to store in the persistent storage managed by the commodity OS.

The number of files that each payment app stores varies with the payment provider. Even though payment transactions are similar at a higher level, application-level differences result in variations of TEE Apps requirements. For example, for AliPay, the TEE App needs to fill in the payment amount, transaction number, payment subject (a title of the transaction) and a notification URL to which the payment server should send the payment confirmation message. In comparison, for PayPal, the TEE App needs to fill in the type of transaction, account number of the user, payment amount and expiration time of the payment. We illustrate the full payment system framework in Figure 4.6 with numbered steps of a payment transaction.

① User initiates the payment and submits all transaction details. In this step, we assume that the user can rely on secure input and display.

② The payment TEE App asks Mint-TEE for an attestation with which it can authenticate its payment request to the remote payment server. Then the TEE App passes a hash of the payment request that includes payment amount, payee and any other payment service-specific values. Note that the payment request itself contains a nonce issued by the remote server to protect the transaction from replay attacks. Since the hash of the request is included in the identity attestation, the attestation itself is also unique per transaction and cannot be replayed.

③ Mint-TEE takes the hash of the TEE App, User-ID and the hash of the payment request provided by the TEE App and generates an identity attestation. This attestation is returned back to the TEE App, which first asks Mint-TEE to store a transaction record on its persistent storage, then sends it to the proxy server via the commodity OS. TEE App and the proxy server communicate after establishing a TLS connection. Because messages are signed and encrypted, the commodity OS does not need to be trusted, and can be thought of as an untrusted network hop.

Besides the network communication, the commodity OS is also delegated with the request from the payment TEE App to write the transaction request record to the persistent storage file, which is also protected by Mint-TEE with freshness and liveness guarantees. To securely keep transaction records on an external disk for persistent storage, the TEE payment app would call Mint-TEE to write to a file for it. Mint-TEE receives the request and forwards the encrypted file with virtual counter value binding to the commodity OS, which returns to Mint-TEE for virtual counter value increment after the job is done.

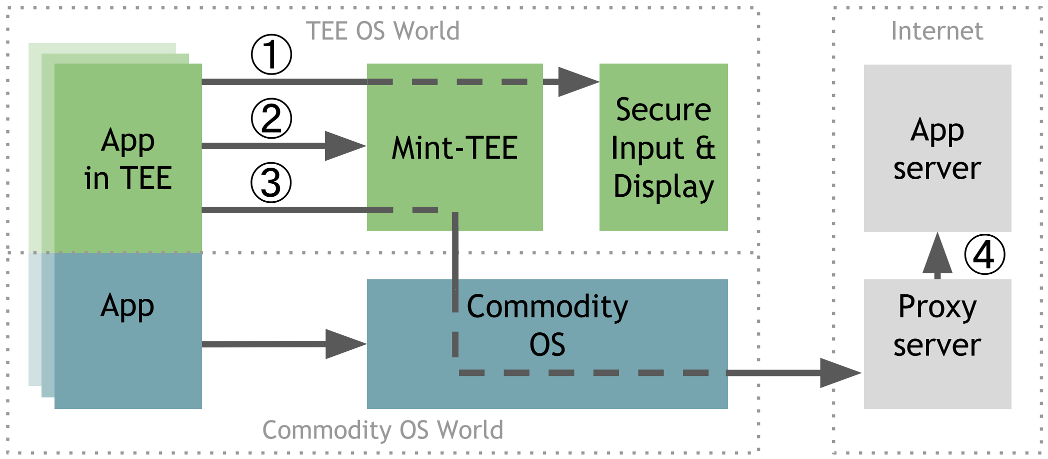


Figure 4.6: System Overview of a TEE application framework in Mint-TEE.

During this process, system crashes may happen either before or after the commodity OS flushes the file to disk, and the states need to be recovered to continue the payment transaction. In the former case, the payment app would need to start over again on this transaction because no real transaction is made and persistent file has not been changed. For the latter case where the crash happens between the flush of the persistent file and Mint-TEE could increment the virtual counter, after the system reboots, the transaction record file needs to be reverted to the previous version with the help of the file system journal. The transaction will also need to be restarted because the payment message has not yet been sent to the server.

④ The proxy payment server receives the signed identity proof of the user, the application and the payment message. It then verifies them and, upon success, forwards a request to the corresponding payment server.

For future work, we intend to implement other types of transactions and payment services. We explain two interesting examples of such transactions:

**Displaying balance confirmation**: In the WeChat wallet service [166], after a payment transaction, the payment server confirms the payment together with a value $b$ indicating the current balance amount of a user wallet. This message can be sent back to the TEE application over the already established TLS connection or a newly established one if the old one has been terminated due to a timeout. The proxy payment service will need to be extended to renegotiate TLS connections for the balance confirmation message.

**Proof of payment**: A payment application (e.g., AliPay) may request the payment server to provide a proof of payment, in a hashed string $p$ which is signed by the payment server, so that the user can later use it to prove to a third-party merchant that a payment/deposit is made. To request the proof, the commodity OS application must collect details such as the payee, date and amount of the transaction and send these to the payment server as a different type of payment request. The payment server can then respond with the proof, which is passed to the TEE application in a way similar to the balance confirmation. However, instead of displaying it to the user, the TEE application will pass the proof to the commodity OS application that requires the proof of payment.

**TEE Secure Chat Application**

We base our implementation of the secure chat application TEE-Chat on the work of Pearl-TEE authors as well. In addition to the original Pearl-TEE version, we add persistent storage protection to TEE-Chat, motivated by existing malware like FinFisher [137] that is used by oppressive regimes to record conversations of political dissidents by infecting and rooting their phones. Our port of TEE-Chat on Mint-TEE uses the Off-The-Record messaging protocol (OTR). The basic framework of the TEE-Chat application is similar to the payment application we described earlier, the main differences are: (1) the chat application needs a symmetric key for encrypting and decrypting messages to be compatible with OTR [131, 12], (2) the chat application needs to interact with the user and ensure confidentiality of its output to the user as well as user's input to the application, and (3) instead of an application server, the end user's TEE application verifies the identity attestation and then displays the attested User ID to the user.

The newly added persistent storage requirement for TEE-Chat is on two types of files: (1) The chat logs, which are multiple files containing messages with one contact for each log file, and (2) a symmetric key for message encryption and decryption. Similar to the cases of payment TEE Apps, the chat log files and the key need to be protected and stored in the commodity OS file system. For both types of files, Mint-TEE needs to work with TEE-Chat to use virtual counters for protecting persistent storage.

The TEE-Chat implementation is originally based on Xabber for Android [191], which is an open-source XMPP client that supports the OTR protocol. In our chat application, we modify the OTR part of the original Xabber application and move all encryption tasks to the TEE application to ensure message confidentiality. In addition, in the secure chat application, users input and read their chat messages via a secure touch screen which is directly controlled by Mint-TEE.

We briefly describe the life-cycle of the secure chat system. While the complete OTR-XMPP protocol with authentication contains more steps, for readability, we elide the details and extract the essential steps for a user to start sending a secure message. We also discuss the possible situation that may happen in the middle of the TEE-Chat process that could affect the persistent storage protection provided by Mint-TEE:

① Mint-TEE attests user and application identity, using the same method as in Sec. 4.4.3 steps ①–③, to the remote end-user. In the `Message` of the identity attestation, the user follows the key exchange procedure according to the OTR protocol and establishes a shared key $K_{SH}$ with the remote chat user.

② The user inputs the chat message via a secure touch screen, and the TEE application uses $K_{SH}$ to encrypt the message, and calls Mint-TEE to store the encrypted message to its persistent storage. The TEE-Chat app will create a virtual counter for the new chat log file and Mint-TEE will update the VCM for mapping the record between the TEE-Chat app and the virtual counter. After it is done, TEE-Chat will hand the message to the commodity OS application for passing to the chat server. Similar to the case of TEE payment application, if a system crash happens between when the file is written to the persistent storage and when the virtual counter is incremented, the chat log file will need to be restored to the latest version after the system resumes.

③ The commodity OS application follows the same procedure as the original Xabber application to generate the encrypted `Message/CPIM` object (the contents of the chat message). It then places the object into the `XML-CDATA` section used in the XMPP protocol and sends the data out to the remote chat user.

④ Upon receiving the encrypted message, the remote end-user application decrypts the message with the shared $K_{SH}$, appends it to the chat log by asking the commodity OS to store the file in the same way, and then composes a response.

In an ideal case of a distributed file storage system, two clients at both sides of a secure chat with end-to-end encryption should have identical copies of chat logs in their persistent storage. However, system crashes may happen at any point during when the messages are being delivered between the two TEE-Chat clients. As discussed earlier in the cases of TEE payment applications, the persistently stored chat log files would need to be reverted to an old version if a crash happens when a chat message is received and written to the persistent storage but the virtual counter has not been incremented. A secure chat log synchronization protocol similar to the Simba [136] can be developed for such cases.

## 4.5 Performance Evaluation

We evaluate the performance of Pearl-TEE and Mint-TEE with our TEE applications. In the evaluations, each TEE application type is compared with a corresponding Baseline application. The Baseline applications are executed on the same system, except outside TrustZone and without the support of Pearl-TEE. We also measure the impact of running a single TEE application using TrustZone versus having multiple different TEE application tenants.

### 4.5.1 Evaluation of Pearl-TEE

| Time Interval | Baseline Applications | | | | | TEE Applications | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | PP | BT | AP | Average | LP% | PP | BT | AP | Average | LP% |
| Local Response | 43 | 45 | 43 | 43.7 | | 121 | 125 | 121 | 122.3 | |
| Proxy Process | 136 | 141 | 145 | 140.7 | 8.4% | 203 | 206 | 225 | 211.3 | 18.5% |
| Payment Process | 305 | 310 | 334 | 316.3 | | 305 | 311 | 328 | 314.7 | |
| Transaction | 495 | 502 | 533 | 510.0 | | 644 | 658 | 683 | 661.7 | |

Table 4.3: Time cost comparisons between TEE applications and their Baseline counterparts (time is in milliseconds). (PP: PayPal, BT: BrainTree, AP: AliPay.)

#### TEE Payment Applications

We implement Baseline applications in Android userspace to compare with TEE payment applications. The Baseline applications are written in C (same as the TEE applications). They communicate with the proxy servers using the same APIs and their behaviors mimic the PayPal, BrainTree and AliPay mobile applications. The only difference is that the Baseline applications do not use any feature provided by TrustZone or Pearl-TEE; the messages sent by the Baseline applications are not encrypted by the SE or Pearl-TEE keys. We measure time intervals that are illustrated in the timeline in Figure 4.7 for each application. The intervals are numbered ① to ⑦

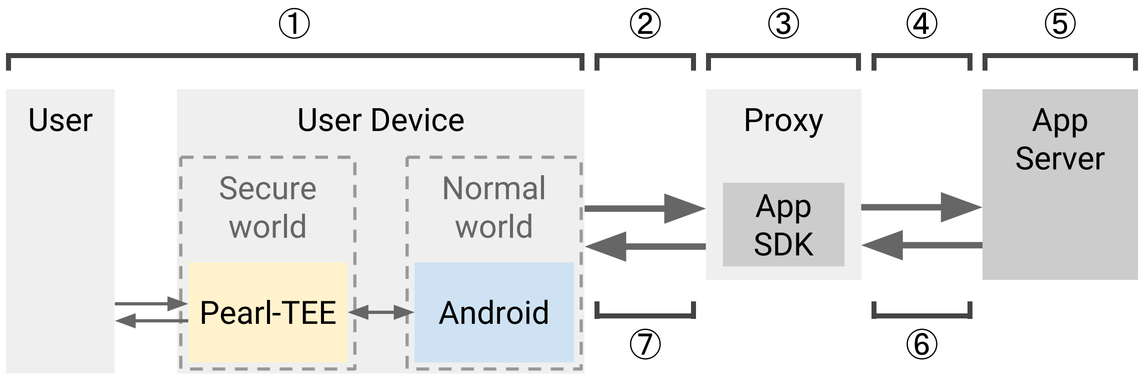


Figure 4.7: The steps used for performance comparison between the Baseline payment application and the TEE-enabled payment application

in the order that payment messages travel through the system. The timing measurements we took are as follows:

**Local response time**: the total time spent on the local user device from the payment button click to the first payment message sent out to the proxy server, which is indicated in the figure by the interval ① (in the Baseline application case, only the normal-world part of the user device is involved). This measures the time that the on-device payment system takes to process local services and prepare the encrypted message to the server. This evaluates the overhead introduced by Pearl-TEE in processing a user's payment request.

**Proxy processing time**: the intervals ②, ③, and ⑦ indicate the time that the proxy server takes to process requests from the user device. This includes encryption and decryption, verification of hashes and processing of payment service SDKs, as well as the message travel time, which includes the time taken to propagate through the network stacks of the device and the server as well as the time taken to execute the protocol handshake. In order to measure the complete transaction time locally on the device, we add a plaintext response message that gets sent ⑦ from the proxy server back to the local payment application on the device.

**Payment server processing time**: the round-trip time between the proxy server and mobile payment service providers. WAN delay and providers' processing times cannot be controlled in our experiment and can be affected by many factors in the network. The payment server processing time includes intervals ④, ⑤, and ⑥.

**Transaction completion time**: the time between the payment button click and the moment when the user receives the confirmation of successful payment on their screen. We measure the total end-to-end delay that a user experiences during the payment being processed (i.e., steps ① to ⑦).

We set up the payment server and development board in the same LAN with a TP-Link AC1350 router of 450Mbps WiFi and 100Mbps wired bandwidth. The payment server runs on a Linux desktop machine with a wired connection to the router. The board we run Pearl-TEE on has an 802.11n wireless connection to the router. The LAN delay between the board and the desktop machine is less than 1ms.

We conducted 20 payment transactions on the TEE applications and the Baseline application, and summarized the average time taken for these transactions in Table 4.3. We note that more

Table 4.4: Performance overhead evaluation for TEE-Chat (time is in millisecond)

| | **TEE-Chat** | **Xabber** |
|---|---|---|
| **Local processing time** | 183.75 | 103.75 |
| **Message round-trip time** | 717.54 | 715.95 |
| **Overhead** | 81.59 | |
| **Overhead %** | 9.95% | |

than half of the time is spent communicating with the payment providers' servers (④+⑤+⑥). The next major component is the network delay and processing time of our proxy server, which could be reduced if it were directly integrated into the payment providers' servers. The SDK processing time on the proxy server is 29.0 ms on average for the Baseline application and 30.3 ms for the TEE applications. We also estimate how much local response time contributes to the overall transaction time with Local Process Percentage (LP%) calculated as (Local Response Time / Transaction Time ) $\times$ 100%.

We believe the total transaction time of less than 700 ms is tolerable in real payment scenarios, even given a worse network condition on a cellular network and a real server. A short delay within a second is still acceptable. Proxy server programs can be further optimized by re-writing them in C.

### TEE Secure Chat Application

To evaluate our TEE secure chat application, we compare the performance of TEE-Chat with the original baseline Xabber application. We measure TEE-Chat performance by running it on our development board and using it to talk to an Xabber application on a Google Pixel phone that has the standard Xabber application running on it. The baseline Xabber measurement is conducted between a vanilla Xabber application running on our development board and the same application running on a Pixel phone. To eliminate the human-operation factor of inputting a message, we modified the Xabber application running on the phone to automatically respond with a message "Gotcha" as soon as it receives a chat message from the application on the development board.

We measure the following values for comparison between TEE-Chat and Xabber:

**Local processing time**: the time between when the user clicks the send button and the message is sent out via the network. For TEE-Chat on Pearl-TEE, the local processing time includes the time it takes to encrypt messages.

**Message round-trip time**: the time between when a message is sent from Pearl-TEE to when a response is received from the phone. The message round-trip time includes the time that packets travel in the network and the remote chat application processing time.

We measure the average of 10 samples and tabulate the results in Table 4.4. We can see that the TEE-chat application on Pearl-TEE only introduces an average overhead of 81 ms, which is less than 10% of the overall message sending/receiving round-trip time. We expect that in a real environment, a user would hardly notice the difference.

Table 4.5: Secure payment system performance breakdown

| System Module | Performance Overhead |
|---|---|
| Network cost & server waiting | 59.3 % |
| Normal-world process | 13.8 % |
| Payment server process | 12.8 % |
| Pearl-TEE kernel operations | 8.2 % |
| TEE application process | 6.5 % |
| World-switching time | 0.1 % |

### Performance Measurement Breakdown

We study the performance overhead of the Pearl-TEE system with a measurement of a few representative events in our system. In this case, we use the TEE payment application as an example. We collect the time consumed by each system component by instrumenting instructions in the entry and exit functions in each component. These instructions use/call the ARM Performance Monitor Unit (PMU), which guarantees the relative accuracy of the timing data. In Table 4.5, we can see that most of the system execution time consists of running processes outside the device, including server processing time, network communication time between the device and the proxy server, as well as between the proxy server and the payment servers.

### Batch Time-out Evaluation

Pearl-TEE enforces a non-restartable batch scheduling policy that terminates non-interactive applications after a fairly short time to ensure fairness. Here, we evaluate what settings for the time-out value $\mathrm{T}_{\mathrm{batch}}$ might be reasonable using the payment applications we have implemented, since they have a significant non-interactive component after the user submits the payment details. If $\mathrm{T}_{\mathrm{batch}}$ is set too small, some transactions might not be able to complete. However, if $\mathrm{T}_{\mathrm{batch}}$ is too large, then the user experiences a longer delay in the event of a network disconnection or server failure. In this experiment, we vary the $\mathrm{T}_{\mathrm{batch}}$ from 350 ms to 850 ms and test each configuration 20 times for the same TEE application (Alipay in this experiment), then collect the triggering rate of the force-termination event.

Figure 4.8 shows that the force-termination is not triggered when $\mathrm{T}_{\mathrm{batch}}$>780ms in more than 99% of cases, which is closely aligned with the message round-trip time plus some additional time for computation. We believe that, with optimizations to the payment server, the value of $\mathrm{T}_{\mathrm{batch}}$ can be further decreased in practice.

### Memory Overhead

We separately evaluate the memory overhead of each module from our implementation in Table 4.6, including TEE payment applications and TEE-chat. We measure the runtime RAM costs of each listed system component. Note that the overall usage of runtime memory cost compared to the host Android OS is negligible since mobile devices nowadays have typically more than 2GB of memory.

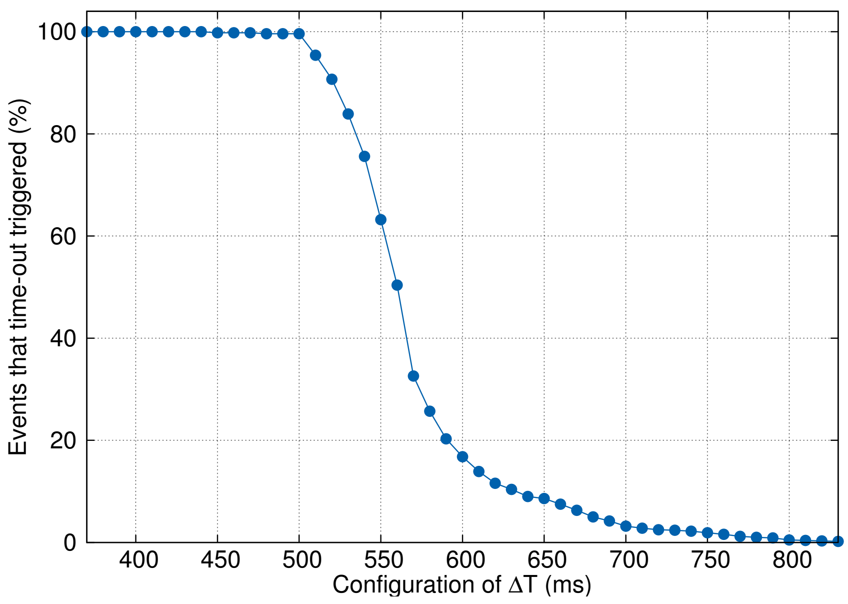


Figure 4.8: Batch scheduling settings and force-termination trigger rate

Table 4.6: Memory usage statistics in Pearl-TEE System

| **System Module** | Runtime Memory Usage (KB) |
|---|---|
| Pearl-TEE | 520 |
| TEE payment application | 9 |
| TEE chat application | 5 |
| Normal-world kernel module | 20 |
| Normal-world user module | 25 |
| Proxy server for payment | 340 |

### 4.5.2 Evaluation of Mint-TEE

We evaluate four aspects of Mint-TEE: (1) performance introduced by Mint-TEE, (2) reduction in the wear rate of hardware counters due to aggregation, (3) increase in the TCB size due to Pearl-TEE and Mint-TEE functionality and (4) the overhead in memory consumption of Mint-TEE and TEE Apps. In the evaluation, each TEE application type is compared with a corresponding Baseline application. The Baseline applications are executed on the same system, except that they are outside TrustZone. We also evaluate the performance impact of Mint-TEE and monotonic counters to defend against rollback attacks.

#### Performance Evaluation with TEE Apps

We set up the payment proxy server and development board in the same LAN with a TP-Link AC1350 router of 450Mbps WiFi and 100Mbps wired bandwidth. The payment server runs on a Linux desktop machine with a wired connection to the router. The board we run Mint-TEE on has an 802.11n wireless connection to the router. The LAN delay between the board and the desktop machine is less than 1ms.

For an evaluation of the effect of virtual counters for protecting persistent storage, we compare

Table 4.7: Performance overhead for TEE Apps using virtual counters in comparison with the same TEE Apps without virtual counters (time in seconds)

| **Task** | $\mathbf{T}_{Baseline}$ | $\mathbf{T}_{Counter}$ | **Overhead** | $\mathbf{T}_{BaselineLocal}$ | $\mathbf{T}_{CounterLocal}$ | **Overhead** |
|---|---|---|---|---|---|---|
| **TEE-PayPal** | 120.13 | 121.41 | 1.07% | 42.36 | 43.54 | 2.79% |
| **TEE-BrainTree** | 122.46 | 123.90 | 1.18% | 43.22 | 44.48 | 2.92% |
| **TEE-AliPay** | 122.17 | 124.32 | 1.73% | 44.80 | 46.16 | 3.04% |
| **TEE-Chat** | 60.00 | 63.20 | 5.33% | 42.62 | 45.15 | 5.96% |

the TEE Apps with a set of baseline apps which do not use virtual counters. We measure the performance overheads by looping the TEE Apps and baseline apps and highlight the results in Table 4.7. In the experiments, we compare the baseline run times of our TEE Apps not using virtual counters with those of the same jobs using virtual counters. For payment applications including PayPal, BrainTree and AliPay, the Baseline apps continuously make pre-defined payment transactions, which are recorded as $T_{Baseline}$. Under the same settings, these payment applications run the same transactions, but with Mint-TEE virtual counters and record the time $T_{Counter}$ for task completion. For TEE-Chat, we generate a random chat message every 5 seconds and send it to a remote client which is also sending messages back. To provide persistent storage state protection, the following counter increments need to be done:

- Before TEE payment app sends out a transaction request, a log of this request is stored in a record file and the virtual counter for that file is incremented.
- When the local secret key of the TEE payment app needs to be updated, its file in persistent storage is updated and the corresponding virtual counter is incremented.
- For TEE-Chat, when the chat client receives a message and writes it to the chat log file on persistent storage, Mint-TEE will increment the virtual counter bound to the file. Similarly, before the chat app sends out a message, the message record file is also updated and its virtual counter is incremented.
- When the symmetric key file of TEE-Chat expires and gets updated to a new key, the file is written to and its corresponding virtual counter is incremented.

In our evaluation for performance overhead, we consider the cases of virtual counter increments due to TEE payment app transaction record updates and TEE-Chat message log file updates. In our evaluation time scale, the key file updates are not expected to happen very frequently. We also take statistics of how much execution time of the TEE App is spent on the local device for baseline cases ($T_{BaselineLocal}$) and for the cases with virtual counters ($T_{CounterLocal}$). Hardware monotonic counter emulation is done by adding empty cycles of approximately 0.1 milliseconds on each access to one such counter and for SE seal operations. The results from real-world applications show that the overhead of payment TEE Apps due to the use of virtual counters is around 3%, and for the TEE-Chat app, which might have more frequent persistent storage access, is below 6%.

We implement another set of baseline apps in the Android userspace, to compare with TEE applications for an end-to-end performance evaluation. For payment TEE Apps, the Baseline apps are written in C (same as the TEE apps) and they communicate with the proxy servers using the same APIs and behaviors as the PayPal, BrainTree and AliPay mobile applications. The only

Table 4.8: Performance overhead for overall protection (time in seconds)

| Task | Baseline | TEE Apps | Overhead |
|---|---|---|---|
| **PayPal** | 103.4 | 112.7 | 8.99% |
| **BrainTree** | 144.3 | 155.2 | 7.55% |
| **AliPay** | 172.6 | 191.3 | 10.83% |
| **TEE-Chat** | 109.6 | 114.8 | 4.74% |

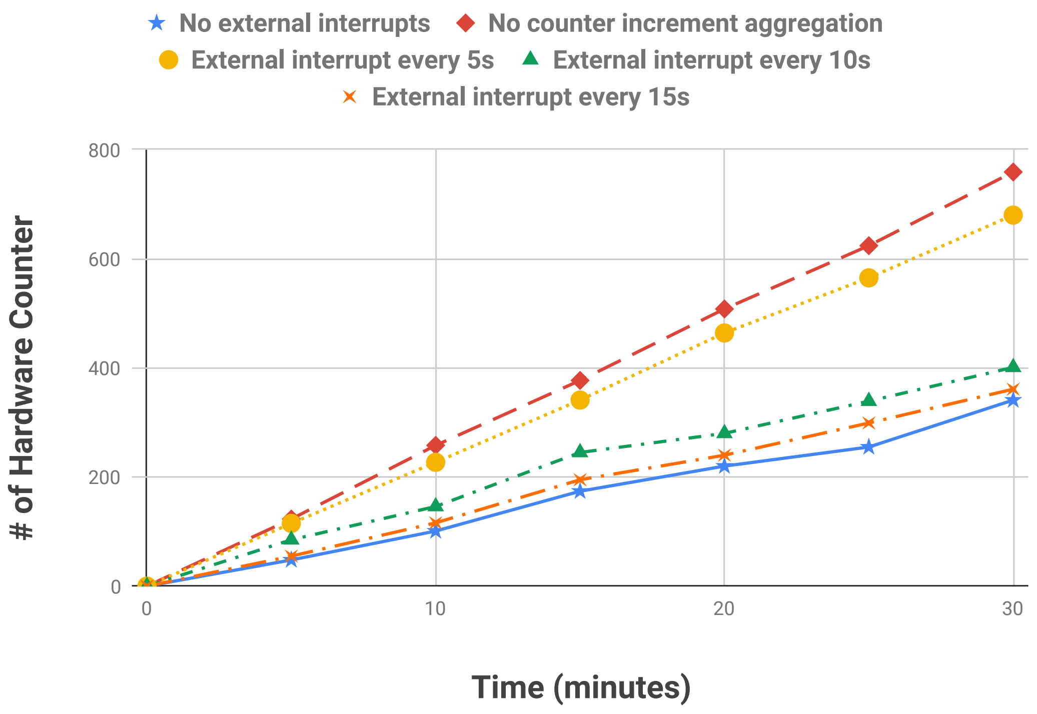


Figure 4.9: Evaluation of Hardware Monotonic Counter Wear-Out

difference is that the baseline apps do not use any feature provided by TrustZone; the messages sent by the baseline apps are not encrypted by the SE or Mint-TEEkeys. For the TEE-chat app, the baseline app we use is the original Xabber application and we measure TEE-Chat's performance by running it on our development board and using it to talk to a standard Xabber application running on a Google Pixel phone. To avoid having to manually input messages, we modify the standard Xabber application to automatically respond with a message "Gotcha" as soon as it receives a chat message from the application on the development board. The results are shown in Table 4.8. The numbers compare the execution times of baseline apps for a set of predefined tasks and the times for TEE Apps execution overall, which gives a quantitative analysis of how much Mint-TEE costs for all the security protection it provides. The end-to-end performance overheads of all our evaluated TEE Apps are below 11%, with an average of 8%.

### Hardware Monotonic Counter Wear Reduction

Mint-TEE uses counter increment aggregation to reduce hardware counter wear. To evaluate its effectiveness, we compare the number of virtual counter increments, which would each become hardware counter increments if Mint-TEE's aggregation were not use, with the actual number of hardware counter increments Mint-TEE performs.

A key difficulty in performing this evaluation fairly is estimating the number and frequency of virtual counter increments that typical TEE Apps are likely to perform under regular usage. Since no real TEE Apps currently exist, we make a best effort by using the 3 payment apps and the secure TEE-chat app we have ported (described in Section 4.4.3). We run them simultaneously by executing pre-collected traces for 30 minutes. While this is only a small number of TEE Apps, we feel this workload can give an indication of the type of counter wear reductions that can be expected.

Another factor that can affect the wear rate savings is the frequency of interrupts that the commodity OS must service. An interrupt that arrives while a TEE App is executing results in a transition from Mint-TEE to the commodity OS, and consequently a hardware counter increment (if there have been any virtual counter increments). As a result, more frequent external interrupts should result in a lower reduction in counter wear due to aggregation. We evaluate this effect by periodically issuing `FIQ` interrupts from the commodity OS every 5, 10 and 15 seconds as well as no external interrupts, and then immediately returning control to the Mint-TEE.

TEE-Chat uses multiple persistent storage files for chat logs, one for each contact in a chat. We set the number of contacts in our experiment to be 6, which means that there are 6 clients of the chat app sending messages to TEE-chat. The message arrivals follow a normal distribution with an average rate of 20 seconds per message of each client. The chat server batches up the messages and send them to TEE-chat every 15 seconds. The traces of the payment TEE Apps are generated by collecting file accesses under manually created payment requests. To make the real-world scenario realistic, one of the authors keeps making small payment requests consecutively using the three payment apps, one after each successful payment.

The numbers of counter increments over time are shown in Figure 4.9. As evident from our experiment, the wear-out rate can be reduced by as much as 55% with no external interrupts. The interrupts from commodity OS affect the aggregation rate as expected: we observe from the results that if secure applications are not getting too frequently interrupted, the hardware monotonic counter wear-out can still be greatly reduced. To optimize the system in the future, we can design the scheduling strategy to let a CPU core be dedicated to a secure TEE App in order to further reduce wear-out of hardware counters.

### Impact on TCB

Maintaining a small TCB for Mint-TEE is critical as it must have a higher level of assurance than the commodity OS to confer any security benefit to TEE Apps. We list the lines of code (LOC) of each module we implemented or modified, and compare them with the original secure OS OP-TEE which Pearl-TEE is also based on. In implementing Mint-TEE on top of Pearl-TEE, we simultaneously ported Pearl-TEE from OP-TEE v2.0 to v3.0. As a result, it is difficult to fairly separate LOC that belong solely to Mint-TEE and those that belong to Pearl-TEE. As a result, we conservatively count all lines of code added to the base OP-TEE v3.0 system against Mint-TEE.

Table 4.9: Lines of code (LOC) of Different System Modules in the Implementation and Comparison with Other OSs

| **Category** | **System Unit Name** | **LOC** |
|---|---|---|
| Server | Proxy Server | 4,500 |
| Commodity OS | Android AOSP | 25,000,000 |
| | Commodity OS user module | 1,800 |
| | Commodity OS kernel module | 1,500 |
| | User application – Payment | 3,000 |
| | User application – Chat | 50,000 |
| TCB | OP-TEE | 159,000 |
| | Mint-TEE | 162,720 |
| | Added libraries (C) | 2,500 |
| | Change to base TEE OS (C) | 1,100 |
| | Change to base TEE OS (ASM) | 120 |
| TEE User Space | TEE payment application | 950 |
| | TEE-Chat | 270 |

Table 4.9 compares the TCB sizes of Mint-TEE, OP-TEE as well as a typical commodity OS such as Android AOSP for reference. As we can see, the base OP-TEE is several orders of magnitude smaller than Android, and Mint-TEE only adds roughly 3,000 LOC to the OP-TEE for a 2.4% increase in TCB size.

The 950 LOC for the TEE payment applications, 270 LOC for the TEE secure chat application, and the $3,000$ LOC for the commodity OS application components represent the total amount of application code for all three payment services in aggregate. We have made necessary changes to the base TEE OS (Pearl-TEE and OP-TEE) by modifying the OS, ARM Trusted firmware and adding new libraries such as the hash tree library to the OS kernel. As a part of the added libraries, the data structure of our hash tree and VCM takes approximately $1,700$ LOC including the Merkle tree library that takes $1,000$ LOC, which we ported from the the secure block device Library [71]. Among the code changed to base TEE OS, virtual counter management and related operations takes up approximately 200 LOC. As can be seen that the majority of the functionality is implemented on the side of the commodity OS.

Finally we show the commodity OS kernel and user modules that implement the message queue as well as the code that persists the virtual counter tree. While these are not part of the TCB, their small size shows that they can be added to a commodity OS with only a modest amount of effort.

### Memory Overhead

We separately evaluate the memory overhead of each module from our implementation in Table 4.10, including TEE payment apps and TEE-chat. We measure the runtime RAM costs of each listed system component. Note that the overall usage of runtime memory cost compared to the host Android OS is negligible since mobile devices nowadays typically have more than 2GB of memory.

## 4.6 Related Work

Providing security features for applications on the mobile platform is not a new problem. One of the pioneering works in the 1990s that has explored and defined interesting problems is the Sanctuary

Table 4.10: Memory usage statistics in Mint-TEE system

| **System Module** | **Runtime Memory (KB)** |
|---|---|
| Mint-TEE | 590 |
| TEE payment application | 11 |
| TEE chat application | 10 |
| Commodity OS kernel module | 20 |
| Commodity OS user module | 25 |
| Proxy server for payment | 340 |

project [199]. The authors raised the question of how mobile applications can be protected from malicious servers and proposed Partial Result Authentication Codes (PRAC) to ensure forward integrity, which means that servers visited later are not able to modify the result of previously visited servers. Trusted execution environments with hardware support were also discussed in the paper to support mobile Java agents running in a secure-coprocessor environment. However, since it was an early work, specific hardware and software platform support was not described.

## 4.6.1 TrustZone Isolation and TEE Application Framework

ARM TrustZone has been employed to design secure systems on mobile platforms [139]. Some work has proposed merging TrustZone with TCG-styled trusted computing concepts to build a trusted computing platform [189]. However, the Linux-based operating system is too large for a TCB to ensure required security. A T&T framework [140] is designed to use NFC combined with TrustZone to preserve a user's privacy during the mobile payment process. AdAttester [102] proposes a mobile advertisement framework that can attest that user clicks on ads are initiated by the real user and displayed ads are intact based on TrustZone. Komodo [51] implements SGX-like enclave protection that has features of attestation and execution isolation with verified software in TrustZone. Sentry [30] is designed to protect sensitive data from memory attacks on DRAM using TrustZone support. However, they have to assume that an attacker cannot compromise the integrity of the associated TEE application, which is complementary to Pearl-TEE.

TLR [150] tries to minimize the software TCB of a .NET implementation and provide an isolation runtime environment for the execution of mobile applications. However, they do not assume a trusted path between the user and TEE application, which limits the domain of applications that can be securely implemented. Moreover, in the use case of secure mobile transactions, TLR also cannot authenticate the user. TrustOTP [164] uses a TrustZone supported one-time password system to provide secure user authentication. However, like much other previous work, they assume that all TEE applications are trusted. MTM [208] is proposed with a mobile trusted module in TrustZone to provide trusted computing for payment applications, TrustPAY [207] is another framework to use TrustZone for mobile payment applications, and similarly, TrustShadow [67] aims at protecting TEE applications on IoT devices. However, none of them consider the case of malicious TEE applications. SeCReT [88] is a framework that enhances the security of the channel between the secure world and normal world, which is complementary to Pearl-TEE and tackles a different problem. Our Pearl-TEE system provides full support for user authentication and secure communication between the user and TEE applications, which protects users from all known phishing attacks. Also, our secure OSes allow multiple mutually distrusting applications to run in the secure domain and protects their integrity.

Our framework provides full support for user authentication and secure communication between user and payment applications. Also, our secure OS Pearl-TEE and Mint-TEE both allow multiple mutually distrusting applications running in the secure domain and protects their integrity along with other security properties.

### 4.6.2 Roll-back Protection

Preventing attacks that aim at rolling back application persistent state and providing liveness in case of power or hardware failure are frequently seen topics. As early as 2006, there is work [151] proposing the use of virtual monotonic counters with the hardware counters on available TPM chips. Memoir [133] uses TPM NVRAM and PCR registers to protect the freshness of applications. However, Memoir requires applications to be deterministic, a restriction that Mint-TEE does not have. ADAM-CS [113] has an asynchronous monotonic service proposed to minimize the maximum vulnerability window. Ariadne [162] extends its previous work ICE [161] to achieve liveness and rollback protection with the assistance of hardware monotonic counters. However, Ariadne requires two monotonic increments for every state update by the TEE App, which would accelerate the wear-out rate of hardware counters. ROTE [114], on the other hand, designs remote counter services to avoid the hardware counter wear-out problem, but the network delay in a real-world setting could be as high as 800 ms. In comparison with the above work, Mint-TEE can protect the persistent states of both TEE Apps and suspended TEE Apps, alleviate the wear-out problem of hardware monotonic counters, scalability issues of the number of supported virtual counters, and achieve a reasonable system delay in our prototype implementation.

Intel's SGX SDK [85] also provides virtual counters backed by hardware monotonic counters. Similar to Mint-TEE, SGX uses a hash tree—however, SGX's hash tree is implemented directly on persistent storage using SQLite while Mint-TEE implements its hash tree in shared memory, allowing the commodity OS to implement its persistent storage for the tree any way it likes. More importantly, Mint-TEE implements increment aggregation and counter aggregation to reduce the wear on the hardware monotonic counters and allow virtual counters to be spread over an arbitrary number of hardware counters.

While these works propose many TEE applications, and often assume the presence of monotonic counters for ensuring the freshness of data, none explicitly discuss how scalability for such monotonic counters can be achieved. One of Mint-TEE's main design goals is to provide scalable freshness and security for TEE Apps.

## 4.7 Summary and Discussions

In this chapter, we have proposed Pearl-TEE, a TEE OS design that enables multiple, mutually distrusting TEE applications that conform to our abstract TEE application model to execute in TrustZone. Pearl-TEE enables and assumes that arbitrary TEE applications will be installed and run in TrustZone and ensures that each application enjoys storage security, execution and memory security, the ability to attest application integrity and user identity to remote services as well as the ability to perform network accesses and access trusted path hardware. The evaluation of our prototype Pearl-TEE system and TEE applications shows insights that this framework can be applied to real-world applications such as mobile payment and secure, privacy-preserving online chat. Our implementation of Pearl-TEE adds less than 3% to the TCB of the TEE OS and imposes less than 20% local processing overhead.

The Mint-TEE we have proposed based on Pearl-TEE, is a secure TEE OS that provides isolation, freshness, liveness and scalability to mutually distrusting applications running in TrustZone. Specifically, Mint-TEE borrows resources from the unstrusted commodity OS and secures those resources with a scalable number of virtual monotonic counters for it and TEE Apps to use. Mint-TEE implements two types of aggregation to reduce the wear rate on hardware monotonic counters and allow multiple hardware counters to be shared across many virtual counters. Our evaluations show that the performance overhead of our prototype can be lower than 10% in average with a non-significant increase of the TCB size of 2.4%.

The lesson we learned from this chapter of Pearl-TEE/Min-TEE is that the initial target audience for a hardware security extension might be relatively small, but as interest and usage expand to a broader range of users, operating system support for adaptations may become necessary to address evolving security requirements. This situation highlights the critical importance of collaboration between hardware and software engineers to ensure that security extensions can be effectively used by the operating systems, while the operating system design should also allow for the continuous adaptation of security applications in response to the dynamic threat and changes of user application requirements.

# Chapter 5

# Aion Attacks: Manipulating Software Timers in Trusted Execution Environment

Side-channel attacks pose a significant threat to secure software operating within a TEE. The TEE is designed to provide a secure environment for applications to execute and be isolated from other processes or from the malicious OS, however, side-channel attacks can hijack and manipulate the OS to probe on the cache and other system resources to form channels that leak sensitive information without needing to break into the TEE. In order to safeguard Intel SGX applications from these attacks, researchers have developed mechanisms to detect cache-probing and frequent interrupts on which these attacks depend. Frequently, these defenses rely on high-resolution timers. However, the absence of a trusted high-resolution timer hardware module has led developers to utilize software timers, which unfortunately underestimate the range of potential attacks. In this chapter, we present Aion attacks [77] that manipulate the speed of a reference software timer to undermine defensive measures against SGX side-channel attacks. Specifically, we introduce a CPU configuration attack that exploits the CPU power management mechanism to alter the execution speed of the timer thread, as well as a cache eviction attack that evicts the target timer counters and compels the system to load them from memory rather than cache. We evaluated the aforementioned Aion attacks and developed an analytical model, demonstrating that software timers cannot be enhanced to accommodate the defenders under our attacks.

## 5.1 Motivation and Contributions of Aion Attacks

In order to protect against side-channel attacks on SGX applications [196, 18, 152, 14, 119, 70, 61, 35, 17, 21], various defensive mechanisms have been proposed [22, 128]. The threat model of Intel SGX assumes that only CPUs are trustworthy; hence, the code and data of secure applications run under protection within a secure enclave, isolated from other system software. While SGX side-channel attacks have yet to be fully resolved, Intel has stated that they are expected and considered to be "a matter of enclave developer" [89]. As a result, software developers bear the responsibility of ensuring enclave security against side-channel attacks.

Interestingly, both SGX side-channel attackers and defenders rely on high-resolution timers for measuring the duration of certain events, such as memory/cache access time, or for counting the occurrence of specific events, like interrupts. However, in the case of defenders, the absence of hardware timers within a secure enclave necessitates the use of software-based timers for high-resolution timing.

Defenders must ensure the accuracy of software timers, leading to careful design considerations on how software timers can precisely emulate real-world time and protect them from attacks. All software timers have some foundational assumptions about their accuracy, which we will explore in detail with our analytical system model. These assumptions can be summarized into two types:

1. The CPU instructions execute at a relatively constant speed, and

2. The clock frequency at which the CPU operates remains within a well-defined range.

For instance, the normal assumptions of the defenders recognize that SGX must defend against an adversary who might modify the processor clock frequency. Thus, they are resilient to an adversary who can slow down the clock frequency by a factor of $3.25-4.25\times$, as this is the typical ratio between the maximum and minimum operating clock frequency of modern processors. In this chapter's work, we show that both of these assumptions can be broken by a significant margin. We present *Aion* attacks, which can be executed by both privileged and unprivileged attackers, and enable an adversary to tamper with software timer accuracy by $2.5 - 202\times$. We also construct a model of software timers and empirically demonstrate that secure software timers are not feasible on current architectures. This renders all current software-timer-based SGX side-channel defenses useless.

Our Aion attacks employ two mechanisms to break the assumptions made by software timers. The first attack manipulates the thermal management facilities of the processor to induce execution slowdown below the lowest supported clock frequency of the processor, violating the assumption that slowdowns are bounded by the lowest clock frequency. To the best of our knowledge, this is the first instance of a security attack that exploits CPU thermal management without physically overheating or damaging the CPU. Instead, we trigger thermal management features using software-only attacks. The second attack generates cache evictions to decelerate the execution of instructions in the software timer, violating the assumption that the execution time of instructions is relatively constant. We demonstrate that these attacks can compromise the security properties of applications running in SGX enclaves and enable existing side-channel attacks to evade detection by existing defenses. We implement a prototype of Aion attacks and evaluate them in a real-world environment. We make the following contributions in this chapter:

- We propose an analytical model which suggests that no existing software timers used in SGX enclaves are reliable, indicating that current SGX side-channel defenses are ineffective if timers are manipulated by attackers.

- We present two generic Aion attacks and demonstrate that they are able to effectively exploit all existing SGX software timers, undermining current defense mechanisms.

- We evaluate two prototypes of Aion attacks on two different CPUs. Experimental results reveal that both consistently break the desired properties of the software timers. With our mechanism, we demonstrate an end-to-end attack where existing side-channel attacks can evade detection.

- We prove that, under our attack model, it is impossible to construct a software timer immune to Aion attacks, thereby motivating the need for hardware support.

## 5.2 System Model

We first propose an empirical model that characterizes software timers through the analysis of both the victim and defender. In this section, we provide detailed descriptions of the attack targets and develop a general model for them.

### 5.2.1 Model of Software Timers

To establish a model of a software timer, we introduce the concept of "'*wall-time*", denoted by $T_w$ hereinafter, as a hypothetical clock that remains accurate and synchronized with the real-world time. We assume that all software timers are designed and implemented to serve the same purpose: tracking the wall-time as precisely as possible and providing the current time when required. Given the lack of dedicated hardware for time indication, we can safely assume that any software timer relies on a sequence of instructions to track wall-time and emulate the behavior of an ordinary clock. In comparison to the wall-time, which reflects the absolute time in reality, a software timer should maintain a "'*clock-time*" $T_c$ and make it accessible to other threads or processes in need of the current time.

In an ideal case, the wall-time is proportional to the clock-time by a constant factor, allowing the clock-time to mimic the wall-time through the execution of a sequence of instructions. The constant factor is determined by the time taken to execute this sequence of instructions. We measure the margin of error from wall-time and define a measure $M_t$ that indicates the accuracy of the software timer in comparison to the absolute time:

$$\frac{|\Delta t_c - \Delta t_w|}{\Delta t_w} < M_t \tag{5.1}$$

In the above form, the clock-times at $t_1$ and $t_2$ are $T_{c1}$ and $T_{c2}$, while the wall-times are $T_{w1}$ and $T_{w2}$, with the elapsed times $\Delta t_c$ and $\Delta t_w$, respectively. $M_t$ is generally assumed to be so small in practice that $\Delta t_c \approx \Delta t_w$, and it is influenced by the execution speed of the sequence of instructions. This speed is affected by both CPU execution speed and memory/cache access speed, so the timer model can account for this by constraining the measure $M_t$.

We can now construct a generalized software timer using the above concepts and employ a global variable $V_G$ to simulate the clock ticks. As different machines possess varying micro-architectures, operation speeds, and cache access speeds, the same timer algorithm will have a variable parameter $I_c$ that reflects the relationship between the variable increase and the clock time. $I_c$ represents the average increase in clock time per value of $V_G$, while $V_{G1}$ and $V_{G2}$ are the values of the global variable at times $t_1$ and $t_2$. With this model, simulating a dedicated timer becomes the problem of using an increasing global variable to indicate the current time, where the parameter $I_c$ allows for general adaptability under the different settings of various machines.

In this manner, when a user or an application needs to measure some $\Delta t_w$ and since $\Delta t_c \approx \Delta t_w$ (previously assumed due to the small $M_t$), they can simply measure $\Delta t_c$ to determine the elapsed time in the clock-time, yielding the result in the following form:

$$\begin{aligned}\Delta t_c &= T_{c2} - T_{c1} \\ &= I_c \cdot (V_{G2} - V_{G1})\end{aligned} \tag{5.2}$$

In this software model, the primary challenge for system developers lies in determining the value of $I_c$. All current software timer approaches assume this value to be relatively stable, as it would result in a small accuracy measure $M_t$. Based on our previous empirical findings, we have:

$$I_c \propto \frac{F_{CPU}}{T_{Inc(V_G)}} \tag{5.3}$$

From the above, the value $I_c$ is proportional to the CPU execution speed, $F_{CPU}$, and inversely proportional to the time $T_{Inc(V_G)}$ required to increment the global variable $V_G$. With these components modeled, we must next examine the victims – the enclave applications and defenders – and how they utilize the software timers.

To summarize, the software timer model measures the accuracy $M_t$ of a software timer by comparing the generated clock time to the wall time. The accuracy is influenced by an intermediate parameter $I_c$, which depends on the execution speed of the CPU ($F_{CPU}$) and the time $T_{Inc(V_G)}$ required to increment a global variable. To slow down a software timer, Equation 5.2 suggests that the adversary should increase $I_c$, causing it to take longer to increment the clock variable by a certain amount. To achieve a larger $I_c$, Equation 5.3 indicates that the adversary may either prolong the execution of instructions or decrease the CPU speed.

With these components modeled, we must next examine the victims – the enclave applications and defenders – and how they utilize the software timers.

### 5.2.2 Defender Model

For the sake of research and without loss of generality, we present an abstracted model of the side-channel attack defenders. This simplified model allows us to analyze and understand the fundamental concepts and mechanisms employed by the defenders while maintaining the applicability of our analysis to a variety of specific defense strategies. The objective of an SGX side-channel defender is to identify attacks. A number of defenses achieve this by measuring the rates or latency of specific events, necessitating the use of a software timer. As an example, we consider two strategies from previous work [22, 128] that utilize the rate or execution time of a measured event as a component of an SGX side-channel defence:

- Cache hit time measurement: The Prime+Probe cache channel attacks on SGX enclaves necessitate the adversary code running on the same physical core as the victim thread [128], as sharing the L1/L2 cache is essential for conducting the probe. One method to prevent this attack involves filling a core entirely with the application's own threads. To verify that two threads share the same physical core, we can measure and compare the time taken by the two threads to access a shared variable in the L1/L2 cache: if it is an L1/L2 cache hit, the access time should be within approximately 10 cycles, indicating shared physical CPU core occupancy. A software timer must be employed for this measurement.

- Asynchronous Enclave Exits (AEXs) Counting: To deploy Prime+Probe cache channel attacks, which grant the adversary a fine-grained cache channel for probing, the adversary actively and frequently preempts the target SGX enclave using, for instance, the high-precision Advanced Configuration and Power Interface (ACPI) or the High Precision Event Timer (HPET). The preemptions trigger an AEX each time they interrupt the victim enclave, which serves as an indicator of side-channel attacks if it occurs too frequently. Defensive mechanisms can count the number of AEX events during a specific time period or measure the time taken by a known sequence of executions [128, 22] and determine whether the rate of AEX events is excessively high to raise an alert for side-channel attacks.

In general, these methods aim to measure the delay or frequency of a specific phenomenon. For example, they may monitor whether there is an *irregular* rate of events $N_{Ev}$ (such as AEXs) occurring during a certain time period. Similarly, the delay of a variable access can be viewed as the inverse of the number of times the variable is accessed within a specific time frame. As a result, all tests fundamentally compare a measured rate of events, $N_{Ev}$, against a threshold, $N_{Th}$, to determine whether an attack is occurring or not in one of the possible scenarios:

$$N_{Ev} > N_{Th} : attack = true \tag{5.4}$$

One dilemma that that defenders face is the choice of threshold: Setting $N_{Th}$ too high will result in missed attacks or false negatives, while setting $N_{Th}$ too low will lead to false alarms or false positives. The typical solution is to establish the threshold based on a calibration run, during which the system is assumed to not be under the influence of an attacker.

Nonetheless, even with this approach, the defense mechanism must account for the fact that the measured rate of events $N_{Ev}$ is dependent on both the true rate of events and the ratio between the wall clock and the rate of increment of the clock variable:

$$N_{Ev} = \frac{\Delta t_w}{\Delta t_c} \cdot R_{Ev} \tag{5.5}$$

where $R_{Ev}$ is the true rate of events according to the wall clock. Normally, we expect that $\Delta t_w \approx \Delta t_c$, so $N_{Ev} \approx R_{Ev}$.

However, recall that $\Delta t_c$ is proportional to $I_c$ in Equation 5.3. Even under benign conditions, there is some variability in $I_c$, which may result in a certain number of false positives and false negatives — typically, $N_{Th}$ is set slightly higher to bias the detection method towards fewer false positives. However, as long as $I_c$ is similar to the value of $I_c$ during calibration when $N_{Th}$ is set, this will constrain $R_{Ev}$ to be roughly equal to $N_{Ev}$. Since $R_{Ev}$ corresponds to how fast an attacker can read a side-channel, constraining $R_{Ev}$ effectively slows the rate of information leakage to the attacker. Nevertheless, if the attacker can arbitrarily increase $I_c$, they can also arbitrarily increase the true rate of events $R_{Ev}$ without being detected. This allows them to probe the side-channel faster and thus reduces the time required for the attack to extract sensitive information from the enclave.

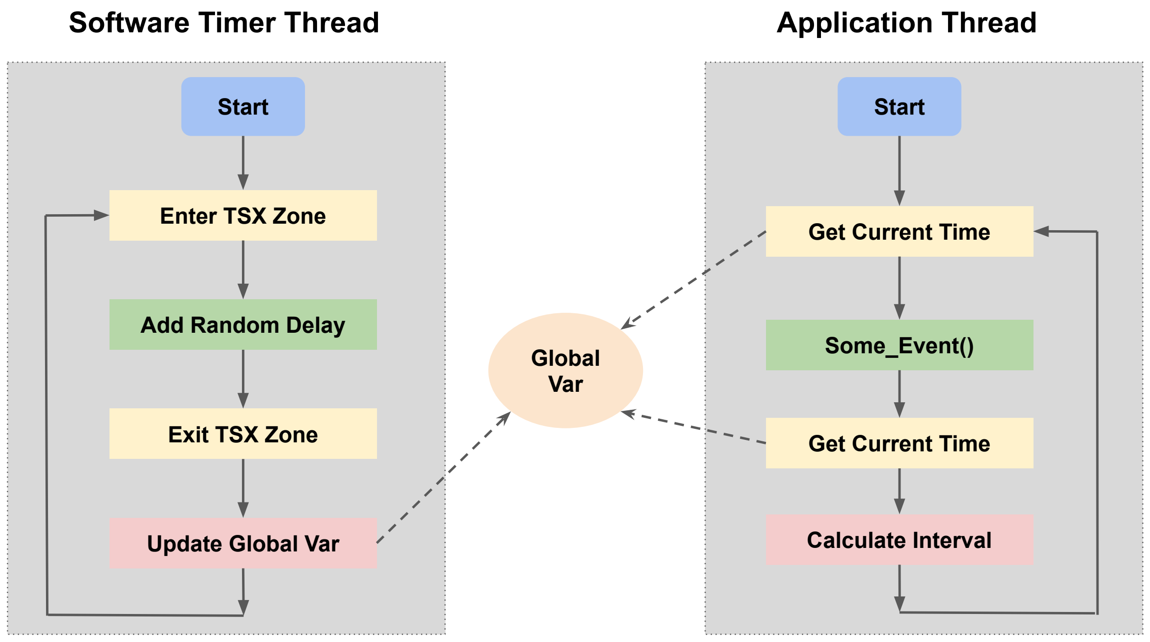


Figure 5.1: A model of how a software timer works and is used by an application thread.

With this general model of a software timer thread, we now discuss why a best-effort software timer design remains vulnerable to attacker manipulation.

### 5.2.3 Timer Countermeasures

We illustrate the general structure of a software timer in Figure 5.1. A software timer thread updates a global clock variable, which is read by application threads to determine the current time.

An attacker aiming to tamper with the timer might attempt to interrupt the timer so as to make the difference between $\Delta t_c$ and $\Delta t_w$ arbitrarily large. To defend against this, both Déjà Vu and Varys employ TSX to detect if the timer has been interrupted. However, TSX can only protect the component of the loop that generates the delay $\Delta t_c$, and cannot safeguard the update of the global clock variable, as the clock variable is concurrently accessed by both the timer thread and application threads. Consequently, the update of the global clock variable must be outside the TSX-protected region.

To prevent an attacker from interrupting and delaying the thread as it updates the global variable, TSX is combined with a randomized delay function inside the TSX region, and the global clock variable is updated with the randomized delay. This makes it difficult for an attacker to predict when the timer thread is outside of the TSX region and can be interrupted without detection. Thus, we summarize that a secure timer, which provides a timing service to other secure application threads, needs to include at least the following components:

1. A global clock variable $V_G$ inside the secure enclave that records the current clock time. The clock time can be read from the clock variable by other threads in the same enclave.

2. A timer loop that increments the clock variable by an amount assumed to be proportional to

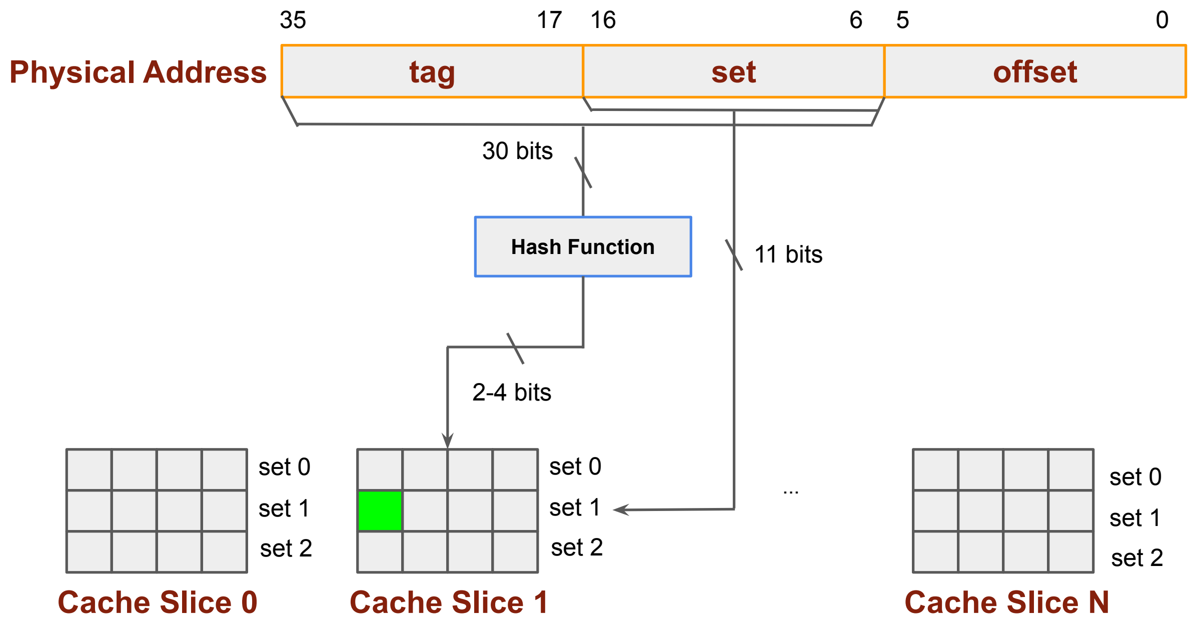


Figure 5.2: Illustration of Intel cache slice and cache set structure.

the elapsed time.

3. A protection mechanism that can either prevent the timer loop from being interrupted or detect if the loop has been interrupted. An example of such a mechanism is TSX.

4. If the entire loop cannot be protected from interruption, a random delay element ensures the attacker cannot predict when the timer is in the unprotected region of execution, i.e., right before the clock variable is incremented.

As we can see, a TSX-protected timer should ideally spend a minimal amount of time outside the TSX region. In other words, the only action taken outside of the TSX region should be to increment the clock variable.

With the added protection of the software timer loop, trivial attacks that manipulate the software timer by frequently interrupting it and/or deprioritizing the timer thread in an OS thread scheduler to make the software timer deviate from the wall clock would not work. Such scheduler attacks that possess OS-level privilege would attempt to preempt the timer thread, requiring an interruption into it. This interruption would be easily caught by the TSX mechanism, and the interrupted transaction would be aborted and detected by the defender.

## 5.3 Attack Design

In this section, we introduce the Aion attacks, which comprise two primary types of exploitation methods. Each of these methods can function independently, and they can also be combined to enhance the effectiveness of the attacks.

### 5.3.1 Aion-1: CPU Thermal and Frequency Attack

The *Aion-1* attack manipulates the rate of increase of the clock variable, indicating the internal time in the software timer thread, i.e., the $F_{CPU}$ in Equation 5.3 of our software timer model in Section 5.2.1. Intuitively, this can be achieved by altering the CPU working frequency through the CPU power management modules of the operating system kernel, causing the clock variable to increase out of sync with the wall time. The strawman method of simply changing CPU frequencies has been described [22], where $procfs$ is utilized to control the CPU frequency from userspace in on-demand mode, and its effect was generally considered to be limited by CPU frequency scaling. Taking the Intel i7-6600U as an example: The processor base frequency (PBF) is 2.6GHz; the maximum turbo frequency (MTF) is 3.4GHz. If the attacker gains control of a CPU power management module, the minimum controllable frequency of a single core is 800MHz. Consequently, it was widely believed that the maximum achievable scale-down of CPU frequency $\Delta t_w/\Delta t_c$ was between 3.25× and 4.25×, a value that most previous defenses could withstand while still preventing an adversary from mounting an effective attack.

However, our attack can challenge the above assumptions using CPU thermal management features. As mentioned in Section 2.5, Intel CPU thermal management is governed by a thermal control circuit, with settings managed by a software adaptive thermal monitor. A process controlled by the attacker with root privileges can trigger a thermal event on the CPU thermal control circuit with instructions that configure MSR registers, for example, `IA32_THERM_INTERRUPT` (0x19B) allows configuration of the thermal interrupt, enabling or disabling various thermal events, and `IA32_TEMPERATURE_TARGET` (0x1A2) allows configuration of the target temperature for the processor, can affect the behavior of the processor's thermal control mechanisms. This not only causes the CPU core frequency and voltage to be throttled down but can also force the processor into the HDC mode, where CPUs are paused for part of the clock duty cycle. Thus, while the clock frequency remains unchanged in HDC mode, the effective execution speed of the CPU is reduced below that of the minimum clock speed, as the CPU is effectively idle for a fraction of the clock cycles. By doing this, we can make the effective execution speed of a CPU approximately equivalent to that of a 100MHz CPU.

According to our software timer model in Section 5.2, the accuracy of a software timer depends on the execution speed of the CPU core on which the software timer thread runs. In side-channel defensive mechanisms, defenders need to ensure the secure enclave occupies both hyper-threads on the same physical core, preventing them from sharing L1/L2 cache with other, potentially malicious threads. They achieve this by measuring the access latency of a shared variable to determine if it hits the L1/L2 cache, since if both threads can hit the cache of the same clock variable within around 10 cycles, they must share the same physical core. To evade detection, the Aion-1 attacker only needs to slow down the software timer, causing the tester to believe that the cache hit time is within 10 cycles during its calibration run, even if it actually hits the LLC and takes around 40 cycles or more, referring to Table 5.1. The attacker can also do the reverse, depending on which thread they wish to slow down. In this way, any secure application that utilizes the software timer will read values inconsistent with the wall time. This deception leads the defender to assume that the variable access has hit in the L1 cache when it could have hit in the L2 or higher.

The effectiveness of the Aion-1 attack depends on the highest and lowest possible processor execution speeds on a single CPU core. The processor execution speed can be considered equivalent

| | Average CPU Cycles | Standard DEV |
|---|---|---|
| L1 Cache | 4.05 | 0.2 |
| L2 Cache | 10.36 | 0.52 |
| LLC | 40.32 | 0.87 |
| DRAM | 152.18 | 1.14 |

Table 5.1: Memory access latencies for different levels of the hierarchy, averaged over 10k accesses and measured in CPU cycles.

to the average core frequency during a specific period. When the attack is being executed, the core on which the software timer thread runs should be set to the lowest possible running speed, while other threads, including the attacker threads, should be set to the highest running speed (or vice versa, when necessary). Without the ability to reliably know its own clock speed, the software timer can unknowingly run slower or faster than the original settings. This type of attack also has some limitations, including:

- The attack can only occur if the attackers are able to access CPU MSRs, necessitating kernel privileges. In SGX applications on multi-tenant cloud scenarios, an attacker might not be able to obtain such privileges, as they would need to compromise the hypervisor.
- The attack also assumes the CPU should support thermal control features, including clock modulation via MSRs to issue a software signal that activates the TCC. Most Intel CPUs available on the market support these features, but not all of them do.

Due to these limitations of privilege and feature availability, we present another attack that employs cache eviction to achieve the goal of manipulating the software timer, potentially as an unprivileged attacker.

### 5.3.2 Aion-2: Cache Eviction Attack

The *Aion-2* attack specifically targets the clock variable utilized in the secure software timer thread through an attacker thread in user space. This is referred to as a cache eviction attack, as it diminishes the execution speed of the reference software clock by evicting the clock variable from the CPU cache to DRAM. In accordance with the software timer model presented in Section 5.2, this attack takes advantage of the stability assumption concerning the cache/memory access speed of the clock variable, namely, the $T_{Inc(V_G)}$ in Equation 5.3.

Intel L1/L2 caches are collectively employed by two logical threads on the same physical core, while all threads utilize the LLC in unison. As the majority of Intel CPUs employ an inclusive cache policy between distinct cache levels, the eviction of the cache line containing the clock variable from the LLC will also result in its removal from the L1 and L2 cache. Consequently, when the software timer thread must increment the clock variable, the thread is required to wait for additional cycles to complete the request, as it must be served from DRAM. Based on our experimental findings, accessing the same cache level necessitates nearly the same number of clock cycles (albeit not the same wall time), and the average DRAM access time is approximately 150 cycles. This implies that the attacker is aware of the extent to which the clock variable's increment can be decelerated by each eviction. For the remainder, the sole responsibility of the attacker threads in user space is to evict the cache line where the clock variable is situated. It is important to note that since the clock variable, which indicates the internal time in the software timer thread, is not safeguarded by TSX transactions, access to the clock variable will not instigate a transaction abort, irrespective of whether it hits cache or DRAM.

In order to execute the attack, the attacker threads necessitate a minimum cache eviction set. A minimum cache eviction set constitutes a collection of virtual addresses that enables a user thread to ensure the targeted cache line is evicted from the cache. For instance, if the virtual address of the clock variable address on a CPU with 4-way associative LLC is `0x00007E30`, the attacker could identify an eviction set of addresses that share the same cache set: `0x00013D30`, `0x00026A30`, `0x000E2730`, `0x0009AB30`. In accordance with the Intel cache structure, this signifies that they should also be located on the same LLC cache slice. The allocation of LLC is relevant because cache entries on different cache slices do not belong to the same cache set. Upon discovering the eviction set, in order to guarantee that the clock variable is evicted from all cache levels, one merely needs to access every address in the eviction set. Subsequently, when the cache entry is successfully evicted, the secure timer thread must access the DRAM to read or write to the clock variable, which in turn hinders the increment speed of the software timer ticks.

We now explain the process of discovering the cache eviction set. As depicted in Figure 5.2, the physical address of each memory request is divided into three components when mapping the address to an LLC cache line. The lowest bits of the address determine the offset within the line, while the set bits establish the cache set to which it is mapped. In contemporary Intel CPUs, the LLC is further segmented into slices, and an undocumented hash function maps the set and tag bits of the addresses to a specific LLC cache slice. Although the hash function itself is undocumented, there have been efforts [115] to reverse-engineer it. Alternatively, other techniques [174] can effectively identify a minimum eviction set using user-level programs with a high degree of probability. We employ existing methodologies for ascertaining an LLC eviction set and utilize them in our attack.

Upon locating a cache eviction set, as illustrated in Figure 5.3, the attacker thread can proceed to access all the virtual addresses in the eviction set of the clock variable within the software timer thread. This ensures that the clock variable is expelled from the cache, significantly reducing the incrementing speed. The attacker can continuously access the eviction set and maintain the eviction of the clock variable in a loop. Consequently, whenever the software timer thread accesses the clock variable again and caches it, the cache entry will be actively evicted by the attacker once more.

As the attacker thread operates concurrently with the software timer thread, eviction of the cache line containing the clock variable is probabilistic, given the absence of knowledge concerning the precise hardware cache replacement algorithm employed by the CPU. Nonetheless, the attacker can enhance the likelihood of cache eviction by parallelizing accesses to the cache eviction set. The attackers can distribute the elements of the cache eviction set among various threads under their control, ideally occupying all remaining available CPU cores with attacker threads. This strategy transforms the single-threaded attack into a multi-threaded coordinated assault, providing the attacker with a higher probability of evicting the target victim cache entry more effectively.

The attacker attains the maximum timer slow-down effect by ideally compelling each increment of the clock variable to miss all cache levels and hit DRAM. Consequently, the software timer operates at a slower pace relative to wall time at the maximum limit, which represents the theoretical worst case for the reference timer. However, due to the inherent complexity of multi-core and scheduler systems, it is challenging for an attacker with only user-level privilege to guarantee this outcome. In

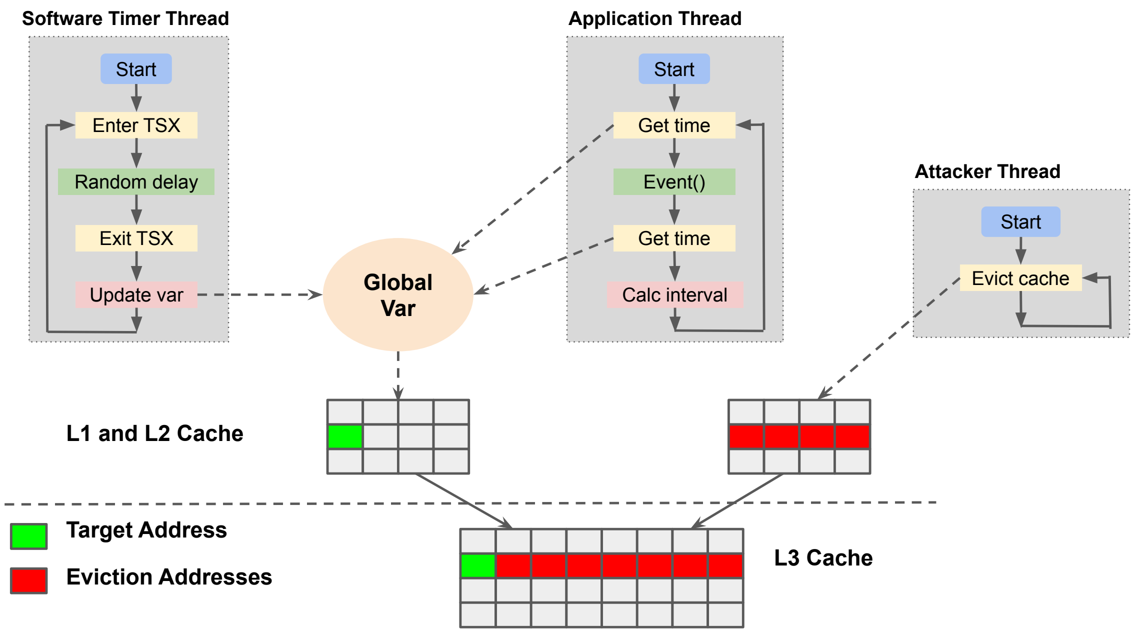


Figure 5.3: Illustration of the Aion-2: Cache eviction attack.

the evaluation section, we will demonstrate the practicality of the attack and present our findings.

In summary, the objective of both variants of the Aion attacks is to manipulate the accuracy of the software timer, either by decelerating or accelerating it without alerting the victim system. According to our software timer model, the attacker's primary focus is on the accuracy measure $M_t$. To compromise $M_t$, the malicious actor may either alter the execution speed of the CPU $F_{CPU}$ or the time $T_{Inc(V_G)}$ required to increase the global clock variable.

## 5.4 Implementation

### 5.4.1 Reference Software Timer

In our experiments on Aion attacks, we employ a real-world software timer as an instantiation of the general model delineated in Section 5.2. We opt for the software timer implementation from Déjà Vu [22] due to two reasons: (1) it exhibits high accuracy for event rate measurement, and (2) it is capable of detecting repeated interruptions and safeguarding itself from malicious preemptions originating from privileged threads. This implementation not only encompasses an essential loop that increments the clock variable, but also additional defensive code utilizing Intel TSX to shield the software timer threads from frequent interruptions by malicious attackers, as demonstrated in Listing 5.1:

- **L**1: The timer thread starts an infinite loop from an SGX enclave.
- **L**2: It enters into a TSX-protected zone, where any interruption to the middle of the TSX zone will fall into a trap, generate an exception, and rollback to the beginning of the entry

```
 1 while (infinite_loop_flag) {
     if (_xbegin() == _XBEGIN_STARTED) {   /* TSX begins */
 3     __asm volatile {
         "rdrand %0\n\t"
 5       :"=r"(rand)
       };
 7     rand = (rand & 0x7) + 1;
       for (i = 0; i < rand; i++) {
 9       for (k = 0; k < 5; k++)
           my_udelay(1);
11     }
       _xend();          /* TSX ends */
13   } else {
       int_flag++;
15   }
     current_time = current_time + rand;
17 }
```

Listing 5.1: Reference Timer Thread Implementation in C

point. This protects the code within the TSX zone, and it can detect if malicious software is trying to interrupt it too frequently.

- **L**3–7: It generates a random integer number between 1 and 8. Here the randomness is provided by `rdrand` as the original authors use it, while other pseudo-random functions could also work.
- **L**8–11: This is a loop creating a delay proportional to the generated random value, so it is harder for attackers to guess when TSX protection covers the thread execution. **L**12 and **L**16: The code leaves the TSX-protected region and the clock variable is updated. As mentioned in Section 5.3, the reason for ending the TSX zone before the timer tick number is updated is that the clock variable is intended to be read from other threads concurrently. If the update is in TSX zone then any concurrent read will abort the transaction and roll back the timer thread.

The clock variable (`current_time` in **L**16) is accessible to both the timer thread and the application threads. The timer thread periodically increments the clock variable with some degree of randomness. When the application thread requires a high-resolution time measurement for a specific event, it first retrieves the clock variable's value prior to the event and reads the same variable again afterward to compute the interval. In this standard procedure, TSX protection does not apply to the clock variable. Consequently, theoretically, anyone can access it without activating the TSX or SGX trap as long as they are within the enclave. However, the timer's accuracy is doubtful for two reasons. Firstly, the thread execution speed is connected to the CPU clock speed because the real-world time of instructions processed by the CPU is dependent on a clock speed not known or controlled by enclave applications. Secondly, the time taken to access the clock variable, whether from the timer thread or any other thread, cannot be assumed to be constant as there is no guarantee regarding the cache or memory level it may hit. Thus, the software timer is not as reliable and secure as previously believed, even when operating within an SGX enclave.

We demonstrate two distinct methods employed to disrupt the in-enclave software timer without the necessity of directly infiltrating the enclave. Nevertheless, these methods achieve the objective of manipulating the timer, thereby confounding the defenders. Consequently, the defensive mechanisms against side-channel attacks are rendered ineffective.

### 5.4.2 Implementing Aion-1: CPU Thermal and Frequency Attack

This type of attack concentrates on altering the speed of targeted CPU cores. The two methods for manipulating CPU core speed include triggering thermal events to force a core into HDC mode and directly adjusting CPU frequency via the power control module of the OS. Since HDC mode prevents a CPU core from running for a certain percentage of the time, both methods can be considered equivalent to making a CPU core run at a specific frequency, which we later refer to as the "equivalent frequency" in this section.

The thermal management attack, as illustrated in Algorithm 1, requires implementation within a thread with root privilege. We trigger thermal events by modifying the respective MSRs: `IA32_CLOCK_MODULATION` and operating on the programmable bits of [3:1]. For direct frequency adjustment, three kernel modules can be used for scaling the CPU core frequencies: `intel_pstate`, `acpi-cpufreq`, and `p4-clockmod`. After testing all of them, we discovered that:

1. `intel_pstate`, as a new power management module, cannot achieve per-core frequency scaling, and

2. `p4-clockmod`, as a relatively last-generation kernel driver, depends on the Intel `speedstep-lib` driver, which is incompatible with our test CPUs. Consequently, we selected `acpi-cpufreq` as the driver that facilitates the attacker thread.

We configure the kernel driver to utilize a "userspace" power governor, allowing a user-level application with root privilege to set any CPU core to run at a specified frequency. In this case, the attacker thread controls the CPU core frequency of the software timer thread and other threads. Various methods can be employed to trigger CPU thermal events, such as configuring the TCC offset of the CPU, increasing CPU workloads to stress the cores, obstructing physical airflow or stopping the case fan from functioning, and sending a software signal to the CPU to enforce clock modulation. We opted for the last approach, which only necessitates a write to an MSR register `0x19A` for implementation. However, we believe that an attacker can employ a range of creative approaches to generate thermal events.

In the attacker thread, we set the target execution speed for the software timer thread to $F_c$ and for other threads to $F_x$. The attacking thread initially obtains information on which CPU core the software timer thread executes and then runs in a loop while `Loop_Flag` is `TRUE` to set the frequencies of the software timer thread and other threads. The attacker thread ceases when `Loop_Flag` is changed to `FALSE`.

### 5.4.3 Implementing Aion-2: Cache Eviction Attack

The cache eviction attack necessitates acquiring the address of the timer variable utilized in the software timer thread in order to identify the cache eviction set. We operate under the assumption that the victim SGX application's image is publicly accessible, which is a reasonable supposition, as

**Algorithm 1:** Aion-1: CPU Thermal and Frequency Attack

```
Data: $F_c$: Target equivalent frequency for core C
$F_x$: Target equivalent frequency for other cores
1  Get enclave application's PID ;
2  Get software timer thread's TID ;
3  Read /proc/PID/task/TID/stat and get CPU core number C that the software timer
     thread runs on;
4  while Loop_Flag do
5      Check CPU frequency $F$ on core $X$ ;
6      if $X == C$ then
7          if $F$ != $F_c$ then
8              Either write to MSR(0x19a) to trigger thermal control of core $C$, or set core
                 frequency to $F_c$ ;
9          end
10     else
11         if $F$ != $F_x$ then
12             Either set core frequency to $F_x$, or do nothing if this is a thermal-only attack ;
13         end
14     end
15     Set X as the number of next CPU core ;
16 end
```

the OS is expected to load it into an SGX enclave. The address can be ascertained through binary analysis of the application image.

Once the address `Addr_t` is found after loading the SGX application, the attack as shown in Algorithm 2 can start to slow down the software timer: First, the attacker thread finds a cache eviction set for `Addr_t`. This can be done by an unprivileged user-level process using a group reduction algorithm [174], or like in our experiment for the ease of implementation, use the page map and get the physical address `Addr_p` of `Addr_t` to find the cache eviction set directly.

As illustrated in Listing 5.2, the undocumented hash function is necessary for determining the LLC cache slice to which a given address belongs, once the physical address is obtained. We acquire the hash function through reverse engineering, employing algorithms from prior work [115]. Subsequently, with the cache eviction set `EV_t` at hand, the attacker thread can continuously access the addresses within the eviction set in a loop, effectively evicting the software timer thread's clock variable from all cache levels. Consequently, this action slows down the timer, as each increment of the clock variable necessitates accessing the DRAM rather than the cache.

We have optimized the attack by parallelizing the attacking loop: the addresses in the eviction set `EV_t` can be further divided and assigned to multiple threads. The attacker threads can keep accessing the addresses in the same eviction set and evicting the target clock variable from the cache faster, because the multithreaded attacker still shares the same LLC and it should take less time for all addresses in `EV_t` to be accessed to evict the target address `Addr_t`.

**Algorithm 2:** Aion-2: Cache Eviction Attack

```
Data: Addr_t: Virtual address of the timer variable in the software timer
      thread
N: Number of attacker threads
1  Find an eviction set EV_t for Addr_t ;
2  if N == 1 then
3      while Loop_Flag do
4          for Addr in EV_t do
5              Read Addr ;
6          end
7      end
8  else
9      Divide EV_t into N sets: EV_1 ... EV_N ;
10     for T in {1 ... N} do
11         Start a new thread Thread_T ;
12         Thread_T loops accessing set EV_T until Loop_Flag == FALSE;
13     end
14 end
```

## 5.5 Evaluation

### 5.5.1 Evaluation Purpose and Experiment Setup

Aion attacks are practical exploitations that can be employed on actual machines. To demonstrate their effectiveness, we assess our implementations of both Aion-1 and Aion-2 attacks in a real-world environment, targeting victim SGX enclave applications. For evaluation purposes, we present end-to-end attacks incorporating a side-channel attack on the SGX enclave, combined with our Aion attacks, in opposition to an implementation of an SGX side-channel attack detector serving as a defense mechanism.

We carry out all experiments on two machines equipped with the following CPUs: (1) Intel i7-6700K featuring 4 cores; and (2) Intel Xeon E3-1230 v6 also with 4 cores. Regarding the system software environment setup, we utilize Intel SGX v2.11 SDK on top of Linux with kernel v5.4, and all machines have hyper-threading enabled. The experiments in this section aim to demonstrate the following aspects:

- Software timers in SGX enclaves are vulnerable to Aion attacks, which can manipulate the reported clock time from outside the enclave.
- Without compromising the software timer of the defender, a representative cache-based side-channel attack will be detected and prevented from exploiting the victim applications.
- With the help of Aion attacks, the same side-channel attacks can evade detection by the defender.

In our experiments, the side-channel attack is implemented to extract an AES key employed for repeated encryptions inside an SGX enclave. We employ the OpenSSL 0.9.7a library, known to be vulnerable to cache timing attacks, as a proof-of-concept demonstration. The effectiveness of our attack demonstration evaluated on the i7-7700K machine illustrated in Table 5.2.

| **# of encryptions** | **Success %** | **Time takes (s)** |
|---|---|---|
| 5000 | 68.7 | 271 |
| 10000 | 90.5 | 520 |
| 100000 | 98.4 | 6534 |

Table 5.2: The Effectiveness of the Base Side-Channel Attacks on SGX Enclave

For the defense, we have implemented an SGX side-channel attack detector based on the Déjà Vu paper [22]. Our experimental results are consistent with the data presented in the original work, demonstrating that it can successfully detect at least 95% of the basic SGX side-channel attacks under carefully adjusted parameters. The results for the i7-7700K machine are shown in Table 5.3. The example test cases we used to run are extracted from the SPEC CINT 2006 benchmarks with the same settings as the base side-channel attacks in Table 5.2.

| **Benchmark** | **Delta** | **Acc %** | **False-positive %** |
|---|---|---|---|
| Numeric sort | 40 | 100 | 40 |
| | **80** | **95** | **3** |
| | 160 | 87 | 2 |
| | 320 | 40 | 0 |
| Fourier | 40 | 100 | 46 |
| | **80** | **96** | **4** |
| | 160 | 74 | 2 |
| | 320 | 30 | 0 |

Table 5.3: The Effectiveness of the SGX Side-Channel Attack Detector

In the subsequent parts of this section, we first present experiments and results illustrating the successful manipulation of software timers in SGX enclaves using Aion attacks. We then demonstrate an end-to-end attack, showing how our attacks can facilitate an existing cache-based side-channel attack on SGX enclaves, effectively evading detection by a defender relying on a software timer.

## 5.5.2 Aion Attack Evaluation

Both types of Aion attacks share the common objective of manipulating the running speed of the software timer, and their effectiveness will be assessed in this section. As previously analyzed, Aion attacks can aid side-channel attackers in evading the detection of existing defensive mechanisms

| | **Xeon E3-1230v6** | | | | **i7-6700K** | | | |
|---|---|---|---|---|---|---|---|---|
| **RF (CT)** | **RD** | **RD+TA** | **TF** | **TF+TA** | **RD** | **RD+TA** | **TF** | **TF+TA** |
| **Base** | **256.3** | 7.0 (37×) | **337.4** | 8.7 (39×) | **225.9** | 5.8 (39×) | **302.5** | 7.9 (38×) |
| **ST (CE)** | 156.2 (1.6×) | 5.2 (49×) | 181.5 (1.9×) | 6.1 (55×) | 148.9 (1.5×) | 4.7 (48×) | 148.1 (2.0×) | 4.1 (73×) |
| **MT (CE)** | 94.3 (2.7×) | 2.3 (111×) | 54.6 (6.2×) | 1.8 (187×) | 90.6 (2.5×) | 1.9 (120×) | 41.7 (7.3×) | 1.5 (202×) |

Table 5.4: Results of software timer readings affected by the Aion attacks. (RF: Randome Function, CT: CPU Thermal, Base: Baseline, ST: Single-Threaded, MT: Multi-Threaded, CE: Cache Eviction.)

that depend on the accuracy of the software timer. We evaluate the degree to which our attacks can accelerate or decelerate the software timer, as this determines the likelihood of a successful side-channel attack.

We combine the two types of Aion attacks and demonstrate their effectiveness in manipulating the software timer as a unit test. We test the reduction rate (the extent to which the attack can slow down the software timer) by the CPU thermal attack and cache eviction attack under various settings.

The baseline workload runs in an SGX enclave as a simple loop that performs operations from AES encryption. We compare the time intervals under different scenarios in Table 5.4, including:

- The baseline scenario where the timer is not under attack, and only affected by the thermal attack at the row of "baseline";
- A single-threaded attacker scenario where only one attacking thread of Aion-2 attack is running; and
- A multi-threaded attackers scenario where the number of Aion-2 attacking threads is the `(# of total hyper threads - 2)`, and the other two threads are taken by the software timer thread and the application thread.

Another variant we compare in the evaluation shown in Table 5.4 is the different random functions used in the software timer thread. Because randomness is not cost-free, all random number generators require varying amounts of execution time, which may impact the time the software timer thread spends in a loop to increment the clock variable. We use four different sets of functions for randomness generation and combine them with or without CPU core execution speed manipulation by the thermal attack:

(1). `RD`: `RDRAND` instruction;

(2). `RD+TA`, which combines CPU thermal attack and uses it under (1)'s settings of `RDRAND` instruction;

(3). `TF`, which is a simple pseudo-random number generator called T-Function [95];

(4). `TF+TA`, which combines the CPU thermal attack and uses it under the same settings of (3) for the evaluation.

We note that since the `T-Function()` is highly efficient in execution, costing less than 30 cycles after optimization, we did not perform a separate evaluation with no randomness generated, which also makes sense in the scenario of real defense. Again, we ensure that the software timer thread, the application thread, and the attacking threads in our evaluation do not share the same core to prevent them from competing for the same processor resources. After each number, the number in brackets (e.g., 202×) indicates its slow-down factor compared to the baseline.

From the results mentioned above, we observe that under various settings, the CPU thermal attack is potent and can achieve a 30-40× slow-down on its own. Furthermore, both single-threaded and multi-threaded attacking methods can effectively decelerate the software timer via the cache eviction attack, and in all but the RD case, achieve a slow down that exceeds the range of slowdowns that previous systems claim to be able to defend. Moreover, when combined with the core frequency manipulation attack, the effectiveness is further enhanced, in total slowing down the software timer by a factor greater than 200 for software timer using `T-function()` under both types of Aion attacks, and by a factor of about 120 for the software timer using `RDRAND` instruction under both types of Aion attacks.

### 5.5.3 End-to-End Attack Evaluation

The previous experiments demonstrated the extent to which the Aion attacks impede the software timer. However, the timer deceleration ratio alone may not sufficiently substantiate that the software timer deceleration rate can effectively facilitate other side-channel attacks in evading detection by SGX side-channel defense mechanisms. Therefore, we have conducted a comprehensive assessment, implementing an end-to-end attack to elucidate the entire process, which amalgamates the conventional SGX side-channel attack with our Aion attacks to circumvent the software timer upon which defenders depend.

The end-to-end attack experiments encompass three components:

(1). A well-established side-channel attack on SGX;

(2). A defense mechanism designed to identify the side-channel attack in (1);

(3). Our Aion attacks, which have the potential to undermine the defensive strategy in (2).

In the experimental setup, the side-channel attack is executed utilizing the SGX-Step framework [172] and employs a Prime+Probe [109] cache-based side-channel attack to retrieve an AES key, which is repeatedly used for encryption operations within an SGX enclave. As a proof-of-concept demonstration, we employ the OpenSSL 0.9.7a library, known to be susceptible to cache timing attacks and compatible with the SGX environment. On our Intel i-7 6700K machine, the success rate of key extractions surpasses 98.4% over 100,000 rounds of victim encryption operations.

For the defensive component, we have examined our implementation of the SGX side-channel attack defender, based on the defense paper [22]. Our findings align with the evaluation data presented in the original work, indicating that the defender successfully detects at least 95% of fundamental SGX side-channel attacks under a trained threshold value $\delta$. This threshold value is obtained and computed by executing standard applications in SGX enclaves without any attacks, thus accounting for and monitoring the typical number of AEX events occurring and initiating the alarm upon detection of an anomalously high quantity of such events. The outcomes within the Intel i7-6700K machine environment are displayed in the Baseline Defence column of Table 5.5, with the `RDRAND` randomness generator employed in the software timer loop.

Distinct from the previous evaluation that demonstrates the impact of Aion attacks on the software timer, the end-to-end attack assessment merges our two variants of Aion attacks with basic side-channel attacks and the SGX side-channel attack defender. We gauge the efficacy of our attacks in supporting the base side-channel attack in evading detection by the defender. Experiments were conducted on two SGX-equipped machines, with the defender employing `RDRAND` as the randomness generator and utilizing both types of Aion attacks combined. The results are presented in Table 5.5. Evidently, under Aion attacks, the defender detects fewer than 2% of side-channel attacks when employing the typical threshold value of 80. When the threshold value is set to 40, the accuracy

| Benchmark | Baseline Defence | | | Defence Under Aion Attack | | |
|---|---|---|---|---|---|---|
| | Threshold | Acc % | FP% | Threshold | Acc % (E3) | Acc % (i7) |
| Numeric sort | 4 | 100 | 97 | 4 | 95 | 94 |
| | 40 | 100 | 40 | 40 | 17 | 15 |
| | **80** | **95** | **3** | **80** | **2** | **2** |
| | 160 | 87 | 2 | 160 | 1 | 0 |
| | 320 | 40 | 0 | 320 | 0 | 0 |
| | 640 | 9 | 0 | - | - | - |
| | 1280 | 3 | 0 | - | - | - |
| Fourier | 4 | 100 | 98 | 4 | 95 | 92 |
| | 40 | 100 | 46 | 40 | 19 | 18 |
| | **80** | **96** | **4** | **80** | **2** | **1** |
| | 160 | 74 | 2 | 160 | 0 | 0 |
| | 320 | 30 | 0 | 320 | 0 | 0 |
| | 640 | 10 | 0 | - | - | - |
| | 1280 | 2 | 0 | - | - | - |

Table 5.5: End-to-end evaluation of Aion attacks against existing defences.

ranges between 15-19%; however, without knowledge of the ongoing Aion attacks, the defender would not opt for a lower threshold value, risking a high false-positive rate. These findings indicate that the combined Aion attacks can effectively aid the base side-channel SGX attack in evading detection by SGX defensive software reliant on a software timer.

In summary, our results reveal that software timer-based defense mechanisms are not feasible in the presence of timer tampering induced by Aion attacks. Comparing the left portion of Table 5.5, which displays false-positive rates under benign conditions at various $E_{th}$ thresholds, with the right portion of Table 5.5, which illustrates detection accuracy following tampering with the Aion attack, it is evident that the detection rate is comparable to the false-positive rate. At the threshold of 80, both false-positive rates under benign conditions are within the range of 3-4%, while the detection rates under attack are between 1-2%. Even upon decreasing the threshold to 40, the detection rates only vary from 14-19%, while the false-positive rates escalate to 40-46%. Consequently, it is implausible for the defender to select a threshold that ensures effective detection while under attack and simultaneously maintains false positives at acceptable levels. This pattern persists across all thresholds. Therefore, our empirical analysis substantiates that employing a software timer in any defense is infeasible due to the adversary's capacity to manipulate the timer.

## 5.6 Related Work

### 5.6.1 General Side-Channel Attacks

We review existing cache-based side-channel attacks from a multi-dimensional point of views, and put them into categories by the techniques they adopt, and discuss how cache noise are discussed or measured in the literature. We will also go through both qualitative and quantitative metrics in our descriptions of the different side-channel attacks throughout this section.

We cover the threat models that the side-channel attacks are using cache-related side channels, in which the micro-architectural channels the attackers rely on is the hardware implementation of an Instruction Set Architecture (ISA). Some works we mention may include micro-architectural features

| Exploitations | Attack Features | Targeted Cache Level | Targeted Crypto-lib | Year of First Publish |
|---|---|---|---|---|
| Prime+Probe | Preemptions in intervals [124] | L1-D | AES | 2006 |
| | Preemptions in intervals [80] | L1-I | RSA | 2007 |
| | Huge Pages [2] | LLC | AES | 2015 |
| | Multi-threading [146] | LLC | DSA | 2018 |
| | Sliding window [16, 116] | LLC | ECDSA | 2009 |
| | Intel SGX [152] | LLC | ECDSA | 2017 |
| | Interrupts across cores [109] | LLC | ElGamal | 2015 |
| Flush+Reload | Page sharing [68, 87] | LLC | AES | 2011 |
| | Sliding window [10, 198] | LLC | RSA | 2014 |
| | Shared library [185] | LLC | ECDSA | 2014 |
| | Protocol responses [86] | LLC | TLS | 2015 |
| | Montgomery ladder [55] | LLC | ECDH | 2017 |
| Flush+Flush | Inclusive cache [63] | LLC | AES | 2016 |
| Prime+Abort | Intel TSX [41] | LLC | AES | 2017 |
| Evict+Time | Lookup Tables [130, 78] | L1-D | AES | 2006 |
| Evict+Reload | Shared library [64] | LLC | AES | 2015 |
| Load+Reload | Cache way predictor [106] | L1-D | AES | 2020 |

Table 5.6: Summary of cache-based side-channel attack examples

as extensions to standard ISA e.g., Intel TSX and SGX. We do not consider the following attacks and threat models: physical side channels including electromagnetic channel, power consumption channel, hardware acoustic channel, and transient execution channel. We also do not include the scopes of network side channels that exploit network protocol features such as RSA paddings. The transparent execution attacks like Spectre and Meltdown are not in our scopes either.

Table 5.6 shows examples of attacks categorized by exploitation techniques and attacking targets. Each specific type of attacks has a target cache level and a target cryptographic algorithm, and they may belong to an exploitation method such as Prime+Probe and Flush+Reload that we will describe later. In the table we briefly describe the core features of each attack and the approximated year of its first occurrence. We next review the attacks in two categorizations by exploitation techniques and target encryption algorithms.

**Flush+Reload**

The Flush+Reload [198, 68, 87, 10, 185, 86, 55] technique is applicable when the attacker shares the same data pages with the victim and is capable of flushing out a selected set of cache lines. To mount this type of side-channel attack, the attacker first flushes a memory line from the cache, and waits for a fixed period of time when the victim may access the line of memory that would cause the line to be fetched into the cache again. After the wait, the attacker accesses the same memory line and times the duration of the access: if the victim had accessed the line, the duration would be much shorter because it is in the cache. Then, the attacker just needs to repeat the above steps and analyze the memory access pattern by the victim, and deduce the secret key. The attacker can use instructions such as `clflush` to target a particular cache line and flush it out of the cache.

The fact that the Flush+Reload method is affected by cache noise was recognized when it was first published [198]. However, they only mentioned that different architecture can have variance in noise level, e.g., some machines are more noisy than others. Other works either mentions the noise could be filtered out by neural networks [68], or indicate that normally noise is observed to be stable in a certain amount [87], while no quantitative results are given on how different level of noise would impact the attacker's capability of extracting keys.

### Prime+Probe

Unlike the Flush+Reload technique, the Prime+Probe [109, 80, 2, 146, 16] method needs to create an eviction set that collides with the victim's cache set, meaning that accessing the addresses in the eviction set would make sure to evict the victim's set from the cache. The prime step is for the attacker to fill in the cache set with the lines from the eviction set. The attacker then waits for an amount of time for the victim to access the memory line that causes a line from the eviction set to be evicted. The attacker continues into the probe step to check in iterations if the victim had any memory line removed from the cache by measuring the cache access time: if the victim had evicted a cache line, the access time would be longer. Through repeating these steps, the memory access pattern of the victim can be analyzed and secret keys can be deduced. The Evict+Time [130, 78] technique is similar to the Prime+Probe in the first phase of prime, then the attacker will wait for the victim executing its thread and evicts cache lines, during this period of time, the attacker measure the timing differences of the victim's execution that reveals which lines were accessed.

Among the above reviewed works, when Prime+Probe was first proposed, the authors discussed some optimization methods on how to measure probe time, and recognized a trade-off between reducing cache noise for attackers and increasing probe time [109]. However, including the original work and the follow-ups [16], they only have limited discussions on how noise can be reduced, but none of them has an elaborated analysis on it can be measured and controlled. One work by Inci `et al.` [80] shows how sensitive the attacks to cache noise by measuring the key recovery effectiveness with presence of cache noise and when the level of noise is controlled comparing the results against a noise-free environment.

### Flush+Flush

The Flush+Flush [63] attack is similar to Flush+Reload in the pre-settings and assumptions about the attacker. For the steps of the exploitation also start from flushing the cache lines shared between the attacker and the victim, and waiting for the victim to access the memory. Then, the second step is to flush the cache line again: because the `clflush` instruction in x86 takes less period of time when the memory line is not in the cache, the second flush would abort early if the victim had accessed the memory line that is brought to the cache again. By recording the period of time that flush instruction takes, the memory access pattern can be analyzed for the use of calculating the secret keys. Comparing with Flush+Reload techniques, the Flush+Flush method is more efficient because the second flush can set up the cache for the next round, and thus it is one-step faster.

The paper mentions that its technique has better noise-resistance in attacking accuracy than other methods like Prime+Probe and Flush+Reload, but the noise level is not quantitatively measured and the performance evaluation results have no controlled group with different levels of cache noise.

### Evict+Reload

The Evict+Reload [64] attack replaces the Flush step in the Flush+Reload technique by eviction. It is a different way of doing Flush+Reload on ARM CPUs, because it does not use flush instructions of x86. Instead, it takes a step to find a eviction set of addresses that can replace victim's memory line out the the cache. The second step is the same as Flush+Reload, and other attacking techniques like Load+Reload [106].

The two papers both mentioned that in comparison with Prime+Probe, they are less susceptible to noise, but how exactly the impact of noise would apply on the attackers are not evaluated.

### Prime+Abort

The Prime+Abort [41] uses features in Intel Transactional Memory (TSX) hardware extension. In the operation, the attacker first primes cache sets and waits for the victim thread to execute and the TSX system will abort automatically because it monitors the cache behavior and when victim thread accesses a critical address, an abort will be sent to the attacker. This technique differentiates from other attacks because it does not have to specifically time cache accesses. The authors of the work have a discussion on countermeasure, where they admit that injecting noise may make such attacks much harder if not impossible, while they also did not present a measurement study with the presence of controlled noise.

All the two or three phases attacks above have the common characteristics. During the different phases, the cache accessing pattern of the attacker and the sensitivity to cache noise might be different. In the common part, they all need to first prepare the target cache set or cache lines by priming or flushing and wait for the victim thread to access the cache, then finally after the victim access, the attackers need to either reload content to cache, flush the cache, or measure the timing of certain operations on cache, or wait for other transactional signals. While between two phases of each attack, the attacker expects the victim thread to access the cache lines of interest, and the cache noise injected here would be treated by the attackers as the victim accesses too. We show the abstracted procedures that used by each type of cache-based side-channel attacks in Fig. 5.4.

## 5.6.2 General Side-channel Detection

A fraction of defensive strategies depends on the results from a detector that is supposed to recognize side-channel attacks. After receiving alerts from side-channel attack detector, the mitigation could work to eliminate threats. There have been various ways to detect a cache-based side-channel attack, through static analysis, trace analysis or runtime dynamic execution.

### Symbolic execution

CaSym [15] establishes two abstract cache models for automatic detection of cache channels using symbolic execution to track both program and cache states, and the symbolic states are sent to SMT solver to decide whether a cache channel exists. Two cache models are used in CaSym: the "infinite model" and the "age model". In both models, a cache state is represented as a set of symbolic memory addresses for variables and array elements, while infinite model has infinite cache size and associativity and age model traces distance of each memory access from the last access. The two abstract cache models are used to model concrete cache replacement policies like LRU and FIFO,

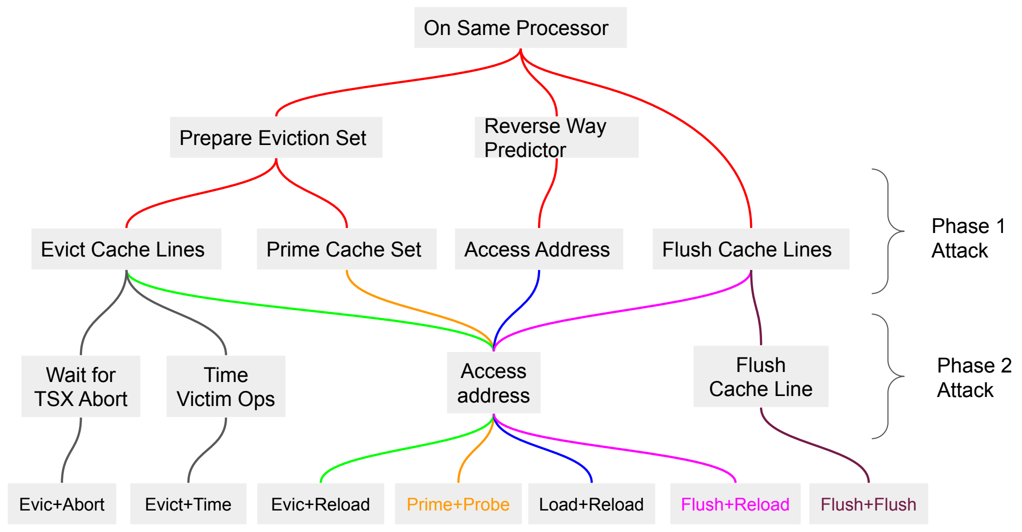


Figure 5.4: Abstracted attacking procedures to categorize different types of techniques used by cache-based side-channel attacks.

and different settings of cache line size and associativity, with different precision and generality of cache channel detection.

CacheD [179] uses symbolic execution towards a single trace to detect side channels. Like all trace-based analysis methods, it underestimates the attack surface and thus has code coverage problems because it only explores one or multiple dynamic execution paths and cannot detect side channels due to conditional branches to confidential data or unexecuted code. In addition, the entire trace of accessed memory addresses should be secret-independent in CacheD's policy, which will cause cryptographic implementations that preloaded keys to be false-positives, because all key-dependent memory will result in cache hits.

### Static analysis

CacheS [178] performs a static analysis on binaries to detect cache-based side channels. It models the secret key-dependent memory accesses and control flow. The model is passed to an SMT solver to detect leakage areas that can be exploited by side channel attacks. CacheAudit [43] is also a tool framework for static analysis of cache side channels. The framework needs program binary and cache configurations as input, and output the analysis on the probability of side channel existence by covering all possible executions.

The research on symbolic execution and static analysis based side-channel detection does not have to consider the impact of cache noises since it normally has nothing to do with them.

### Trace-based dynamic analysis

DATA [186] is a tool to log dynamic execution traces and uses differential analysis for detecting side channels. Stacco [192] and MicroWalk [187] both use dynamic binary instrumentation for detecting side-channel leaks, while the former focuses on the scenarios in Intel SGX, and the latter focuses on general closed-source binaries. Nights-watch [122] uses a machine learning approach to recognize attacks based on information from hardware performance counters.

Either by applying performance monitor or by collecting software execution traces, such methods could be potentially affected by the level of cache noise naturally or deliberately added into the system. However, they have missed such opportunities to make the factor of cache noise accountable and measurable.

### Runtime detector

Some runtime detectors can monitor the CPU performance counters to detect side-channel attacks in real-time (SCADET [148], CloudRadar [205], SpyDetector [100], Chiappetta et al [25]). The issue of false positives is common for all these methods, and the mechanisms here leave the detection results to system administrator for a proper resolution instead of mitigating the attacks by themselves. Each of the techniques closely depends on specific cryptographic algorithm they use. For example, the system of SpyDetector [100] uses a semi-supervised anomaly detection model for detection of general runtime side-channel attacks on AES. SCADET [148], with a similar approach, focuses on tracking a specific type of side-channel attack, i.e., Prime+Probe. CloudRadar [205] tests their runtime detector on IaaS cloud systems using Open Stack as an example, and requires cloud users to provide application signatures for it to decide whether the protected application is running. With the user-provided information, it can achieve a low false positive rate of detection. Chiappetta et al. [25] uses hardware performance counter for side channel identification in real time, the cache hits and misses information is collected by the detector and fed into a machine learning model as a classifier to make the final decision. The model learns on features of cache misses, cache accesses, and the number of processes in the execution windows, and gives a prediction of the workload level. The system should raise an alarm for a possible attack if a window is not within the cluster. For each cryptographic algorithm, a different prediction model should be specifically trained and tied to it. Since each application also requires a different window size, the generality of the prediction model is in question. The high-false positive rate (up to 30%) may also become a main obstacle that prevents the application of such mechanisms.

The judgement of the runtime detector is expected to be affected by cache noise. For example, SCADET [148] mentions that in their experiments, real noise like browsers and network applications could be added, but the level of noise is also not measured. Similarly, other works of runtime detectors do not consider evaluating how much likelihood cache noise would introduce false positives and what the consequences are.

Some scheduler-based mitigation approaches [92, 173] rely on a runtime detector against side-channel attacks. For example, Biscuit [92] detection strategy is based on the statistics that the side-channel attack victim process has at least 5 times the number of cache misses than in the normal execution. The detection of side-channel attacks relies on the prediction of cache misses in loops, which is then sent to a compiler-assisted scheduler that decides how much time share should be given to each process, according to the cache footprint of the loop. The scheduler also relies on the monitor finding which process causes victim to miss cache heavily, and de-prioritize the suspect to mitigate the attack. The mitigation comes with the cost of performance overhead up to 11%. Besides, the scheduler assumes the number of side-channel attacking threads is less than 8.

### 5.6.3 General Side-channel Attack Mitigation

#### hardware-based Mitigation

SHARP [197] proposes a cache replacement policy to defend against cache-based side-channel attacks. The core of the hardware change is to eliminate the inclusive shared cache replacement policy, which means that an attacker should no long evict the probe address to cause victim's cache on other processes also evicted. In SHARP replacement algorithm, the hardware needs to tag a line of cache belonging to "private cache" in order to protect it. The replacement decider is a decision tree with an event counter that keeps the counts of the cache line to be replaced is shared. When the event count reaches certain preset threshold, an interrupt will be generated to prevent the suspected process from running. Note that this new cache replacement method is only tested on simulators. In addition, to prevent side-channel attackers from invoking cache flush instructions, the defender restricts user from executing them. The side-effect of this approach includes user cannot use instruction like `clflush` in their legit use cases when updating memory location.

Wang et al. [182] designs a new hardware cache systems to mitigate cache side-channel attacks. Two approaches have been raised to tackle the cache interference problem: Partition-Locked cache (PLcache) and Random Permutation cache (RPcache). PLcache's strategy is dynamically partitioning the cache that the cache lines of interest are locked in cache as a "private partition", which is not evictable by other cache accesses not belonging to this private partition. In this way, cache interference is eliminated because the hardware can prevent undesired eviction when protected processes put data into a private/locked cache. Some ISA changes are required to perform the lock/unlock operations with new load/store instructions e.g., `ld.lock/ld.unlock`, `st.lock/st.unlock`. The drawback is that by locking up caches, it degrades the cache utilization and consequently affect performance. RPcache's strategy, unlike PLcache, allows cache sharing. It tackles the cache interference problem by randomizing cache accesses, so no useful information about which cache line was evicted can be inferred by the attacker. In RPcache, the hardware keep the mapping between memory and cache randomized, such that the attacker cannot infer critical information about victim process from knowing which cache lines have been accessed.

A follow-up work [181] to the RPcache strategy [182], that it diversifies cache mapping to obfuscates cache accesses. The proposed technique is called dynamic memory-to-cache remapping, which means that a memory line can be mapped to any cache line at run time using a ReMapping Table (RMT). RMT can be continuously updated by cache replacement algorithm every time a cache line replacement occurs.

However, the above two approaches are still vulnerable against alternative cache attacks [97] outside their original threat model, so new patches [98] have been proposed to fix them by introducing techniques such as preloading to secure PLcache with hardware changes, and use software permutation instead of permutation hardware in RPcache for randomizing the cache address maps.

Gruss et al. [62] proposes a method of cache access randomization that defends the type of side-channel attacks that only exploit the reuse of a previously cached or accessed critical data. In the threat model, the attackers do not need to generate any cache contention, they only need to calculate the correlation of the addresses of two memory accesses by analyzing reuse of the caches. The new cache fill policy randomizes the cache fill in the previous default "demand fetch" policy which makes cache fills correlated with demand memory accesses and leaks information about previous memory accesses. The paper shows that randomizing cache fill would fix these problems by de-correlating the relationship between cache fill and demand memory access.

CEASER [141] targets at mitigating conflict-based cache-channel attacks such as Prime+Probe, Evict+Reload by address encryption and remapping. The key inspiration of this technique is combining cache address encryption and randomized mapping, as it finds out that accessing cache with encrypted addresses would be scattered to different sets of cache, which serves the purpose of remapping better. CEASER uses low-latency block-cipher (LLBC) to encrypt physical line-addresses into encrypted line-addresses, and the latter is only visible to the LLC, so that all store and prefetch requests access the cache with only physical line-addresses: the CEASER works like a transparent address converter. The performance overhead based on their pin-based simulator shows under 1% overhead. The vulnerabilities of CEASER is pointed out by Qureshi [142] , for example, replacement policy like LRU can still be exploited under low remap-rate of CEASER. One of the solutions that it proposes to solve this issue is to partition the cache and map cache lines into different sets in partitions.

Sapper [104] proposes a hardware design language "Sapper" that specialized in designing security-critical hardware components by enforcing information flow policy. The language is a state-machine-based language that can support checking possible information leaks due conditional execution and other operations. A compiler that supports synthesizable subset of Verilog is needed for Sapper to insert dynamic checks in the hardware designed. While Sapper being a general HDL, SecVerilog [204] focuses on timing leakage channels as a Verilog extension: it assures timing-sensitive noninterference of information flow, such that cache channels would not exist. SecVerilog has a type system that statically controls information flow and verifiable. The evaluation of SecVerilog is based on MIPS and its cache design, assisted proof by Z3, enforcing timing label contracts to eliminate cache channels. The performance overhead with timing channel protection is claimed to be 12.2% - 21%.

### Software-based Mitigation

CacheBar [209] proposes two techniques to mitigate side-channel attacks on LLC. The first method makes copies of shared pages and keeps them private to prevent processes from sharing sensitive cache lines. In this way, they dedicatedly protect processes that have shared library pages doing cryptography functions from being traced by side-channel attackers to flush their cache lines by applying Flush+Reload or Flush+Flush attacks. The second method simply limits the number of cache lines that a process can access. In this way, Prime+Probe and Evict+Reload can no longer traverse the entire eviction sets, or it cannot record all the cache accesses by the victim to analyze the timing information.

The side effects of above approaches include performance overhead introduced by creating private pages. Also, limiting cache access is not good for performance of benign processes as well, the overhead can be up to 25%.

StealthMem [94] proposes the idea of "stealth pages" which are isolated pages allocated for each process. The stealth pages are mapped to unique cache sets that are not shared with other pages. To protect critical processes such as the ones of cryptographic applications, StealthMem assumes critical data and functions are strictly within these stealth pages, so the source code of the protected applications need to be modified to add explicit constraints. The stealth page method is essentially a software-version approach of cache partitioning, as it also reduces the utility of LLC and the

| Type | Countermeasure Examples | Brief Descriptions |
|---|---|---|
| Isolate/Random Cache | Cache coloring [94, 28, 155] | Allocate cache or pages by coloring |
| | Cache sharing limit [209, 42] | Limit cache accesses of co-located threads |
| | Cache randomization [108] | Randomizing the cache mapping |
| Insert Cache Noise | Düppel [206] | Adding noise by cache cleansing. |
| | Cache Decay [91] | Randomly causing cache entries to decay |
| | Server-side solution [57] | Flushing selected caches lines repeatedly |
| Scheduler-based defence | Biscuit [92] | De-schedule suspicious threads |
| | Scheduler-based defense [173] | Enforcing minimum runtime |
| Transform Program | Raccoon [144] | Obfuscate programs |
| | Software diversity [32] | Auto-generate replicas of programs |

Table 5.7: Summary of side-channel defence

reduction expands with the number of CPU cores. The overhead reported is up to 11% on 4-core CPU with 6 VMs.

Program transformation technique [190] is proposed either to make CPU cycles and cache misses to be independent of critical data like keys, so that the side-channel attackers cannot infer secret information from recording patterns of cache hit and misses, or to make programs execution to cover as many as possible branches so cache accesses pattern should be confused to cache attackers. The program transformation has impact on delay and throughput of running processes, the average overhead is 50% and worst case is up to 225%.

Other similar approaches can make replicas of programs [32] to cover cache access patterns of the victim program and confuse the attackers who wants to probe the cache or measure certain cache operations. Both methods of program transformation can create a noisy cache environment for the attackers to achieve their defensive targets, while the latter gives an extensive and quantitative study on cache noise inserting: The noise is considered to be a composition of “random system noise” and cache noise manually added with an insertion rate. With different cache noise insertion rates, they evaluate the effectiveness against Prime+Probe and Evict+Time attacks.

Dynamic page coloring is another software approach to mitigate side-channel attacks, which also directs how memory pages are mapped to cache lines. Shi et al. [155] ensures that a group of pages with same color will always be mapped to a fixed set of cache lines to prevent undesired cache sharing, however this approach incurs heavy performance overhead up to 2.2x.

Düppel [206] defends virtual machines from cache channel attacks happening on public clouds without needing to change the hypervisors. The method they use is injecting noise from VM OS kernel, so attackers who need to observe the cache timing would fail. The noise that Düppel adds by repeatedly cleaning L1/L2 cache alongside the executing of tenant VM workloads. The performance overhead of Düppel when no attacks happening is less than 7%, and false-positive rate is below 3% (caused by its detector).

Some works use existing hardware features/extensions to protect the system from side-channel attacks: CATalyst [107] uses Intel Cache Allocation Technology (CAT) hardware feature to divide the cache into cache partitions. Because the number of CAT-supported cache partitions is limited, CATalyst further uses software page coloring within the secure cache partition to block interferences. The performance overhead depends on how much cache is reserved for secure cache partitioning: With the case of 10% reserving, according to SPEC an PARSEC benchmarks is in average less than

1%, with 6.5% as the worst case. When larger part of reserved cache partition, the overhead can be up to 6%, and 45% in some worst case.

Cloak [62] uses hardware transactional memory (TSX) features to protect SGX enclaves from cache-channel attacks, by preventing cache misses on critical code. The approach can be used to hide the cache access patterns from attackers, because the TSX can guarantee that in the event of conflicting concurrent memory accesses, the transaction aborts and all corresponding changes are rolled back. The overhead statistics from SGX evaluation is -0.8 – 248% depending on how memory-intense the evaluated jobs run in the enclave.

Varadarajan et. al [173] try to guarantee a principle of "soft isolation" among VM scheduling. The method protects the security properties by enforcing minimum runtime guarantee (MRT) for vCPUs to prevent high-frequency of preemption, and restricting the attacker's capability of conducting enough number of interrupts in limited time. The approach can disrupt the malicious VMs that are trying to record victim VM's cache access patterns. Moreover, to provide soft isolation, the mechanism needs to clean CPU states between preemption, which degradates the performance of the cloud infrastructure and slows down all the VMs running on cloud. Biscuit [92] requires information received from a side-channel attack detector that recognizes which thread is a suspicious malicious one, and limit the execution of the suspect by putting it to the end of scheduling queue. Sprabery et al. [160] makes changed to the Linux default CFS scheduler to help defend side-channel attacks by forcing protected threads and other threads to be temporally isolated, such that the chances are reduced that the attacker can share the same cache with the victim thread.

We finally summarize a subset of the mitigation techniques into Table 5.7, which is a selection of defence that work directly on cache.

### 5.6.4 Cache-Based Side-Channel Attacks and Defence in SGX

Cache-based side-channel attacks on SGX are based on general cache channels like Prime+Probe [124] and Flush+Reload [198]. The basic idea of such attacks is to measure the access times to a series of specific addresses and use the information to infer whether they are in cache or not. By learning the victim's memory access patterns, attackers can retrieve confidential data like private keys.

These attacking methods have been widely applied to exploit SGX applications. Malware Guard Extension [152] develops a Prime+Probe type of attack with the help of a high-resolution timer to distinguish cache hit and miss and uses the LLC cache channel. CacheZoom [119] also falls into the category of Prime+Probe attack while it keeps interrupting the victim enclave and obtains high-resolution information about its cache access. Brasser et al.[14] show the practicality of conducting a cache-channel attack using L1 cache to extract RSA keys of the victim application running in an SGX enclave. Other side-channel attacks[61, 35, 180] are based on similar methods, with a common strategy of probing a high-resolution time of the victim enclave application's cache access and then inferring secret data from the enclave.

Defensive mechanisms against cache-based side-channel attacks also depend on high-resolution timers and have to implement their own software timers inside the SGX enclave when needed. Varys [128] defends against cache-based side-channel attacks by enforcing security-sensitive threads reserved on the same CPU physical cores, allowing it to detect attacker threads attempting to access shared CPU resources. In their design, thread allocation detection depends on a high-resolution software timer inside the SGX enclave. Déjà Vu [22], as another strategy of protection from cache-channel attacks, constructs a more complex software high-resolution timer assisted by Intel TSX. Part of the software timer is protected by TSX for detecting repeated interruptions from the attackers, but not all of the timer thread is under TSX's protection due to software timer use cases.

## 5.7 Summary and Discussions

Side-channel attacks pose significant threats to TEEs, including Intel SGX. While software-based defenses have been proposed for detecting and preventing cache-based side-channel attacks, the absence of a reliable hardware timer for secure applications to utilize within the enclave renders such solutions susceptible to Aion attacks. In this chapter, we devise and implement two variants of Aion attacks: one predicated on manipulating the software timer thread execution speed by inducing CPU thermal events, and the other concentrating on cache eviction to decelerate the rate at which the target timer increases. Both approaches can effectively alter the speed of the software timer in an SGX enclave, thereby invalidating defensive strategies reliant on precise high-resolution software timers. We further contend that, within our general software timer model, devising a dependable timer purely in software to render the defense usable and effective in detecting side-channel attacks is unattainable, unless the defense can tolerate up to a 200x slowdown of their timer, which is improbable.

The crux of the issue, as demonstrated through our model analysis, lies in the erroneous assumption made by system designers when utilizing a software timer to measure the duration of critical events. They presume that the increment speed of the variable employed in the software timer is nearly constant and cannot be significantly altered by an adversary. However, this assumption fails to consider the drastic changes in execution speed that can transpire when accessing a global variable, resulting from variations in CPU frequency or manipulations of cache behavior. By disrupting the high-resolution measurement of time, Aion attacks can exploit existing defence.

The lessons we learned from this chapter of Aion attacks emphasize the importance of meticulously examining hardware features that may initially seem inconsequential but can undermine software defenses using hardware security extensions. This realization highlights the need for thorough testing, analysis, and ongoing research and development to stay ahead of potential security threats. A comprehensive approach, considering both hardware and software aspects and their interactions, is essential when evaluating and designing security solutions.

# Chapter 6

# Discussion and Conclusion

## 6.1 Discussion

From the lessons we have learned through the three pieces of work in this thesis, we have a few topics that were not covered in the previous chapters yet we think worth discussing in the end of this thesis:

**1. Deprecation of Security Hardware Extension**

On-chip resources are finite, thereby necessitating hardware designers to judiciously evaluate and select the functionalities and instructions they intend to support. Consider the Intel MPX case; the hardware extension was initially introduced into the development emulator in 2012, eventually incorporated into CPU production in 2015, and later listed as removed by Intel in 2019. This life cycle of the hardware extension was punctuated with years of code contributions and critical reviews from software developers across different communities, aimed at optimizing operating system kernels, runtime libraries, and compilers.

For instance, within the GNU/Linux open-source community, the journey of MPX support was notable. It was included as early as GCC 5.0 and then phased out in GCC 9.1. Similarly, Linux kernel extended its support for MPX from version 3.19 to 5.6, and the QEMU emulator maintained support for it from version 2.6 to 4.0. The effort encompassed contributions from numerous developers, extending beyond open-source platforms to include proprietary software like Visual Studio and the Intel C++ compiler, and encompassing numerous academic projects.

Though the extension was initially greeted with enthusiasm by the security community, its short three-year lifespan ended up being a disappointing aspect. A key lesson derived from this case study emphasizes that software features developed in conjunction with a specific hardware extension should be planned with a clear vision of their lifecycle, whether short or long. This is due to the inherent unpredictability concerning the feasibility and longevity of the hardware, making such a strategic approach all the more vital.

**2. Trade-Off Between Open and Close TEEs**

Enduring hardware features such as ARM TrustZone present similar challenges to the developer community. As discussed in Chapter 4, most personal mobile devices are shipped with ARM processors supporting TrustZone, and the developer's guide is readily available to the public [56]. However, a significant majority of end users operate mobile systems with closed TEEs, which stands in contrast to the open TEE system proposed in this thesis.

The disadvantages of open TEEs are as apparent as the benefits we described in the Pearl-TEE and Mint-TEE systems. For instance, the attack surface is more or less broadened by allowing third-party applications into the TEE, setting up and maintaining remote attestation services become necessary, and it becomes imperative to educate users to identify phishing attacks designed to trick them into installing malicious TEE apps.

Manufacturers of mobile devices need to contemplate these factors when deciding whether to incorporate an open TEE solution into their products. Therefore, another critical lesson we have learned from this study is the ongoing tension between openness and closeness. This dichotomy invariably influences software feature development and manufacturers' decisions regarding secure operating systems. It underscores the inherent trade-offs to consider in the pursuit of security and flexibility for users, and how these impact both software development and hardware manufacturing decisions.

**3. Responsibility for Hardware-Related Security Vulnerability**

As detailed in the Aion attack background in chapter 5, it's evident that cache-based side-channel attacks can potentially be addressed by either software defense measures or hardware modifications. Intel provides a secure coding guidance [82] for developers, which recommends practices such as constant time code and compiler optimization to mitigate side-channel attacks. Conversely, software developers are advocating for additional hardware support to tackle these issues.

Our Aion attack case study elucidates this dilemma. The lack of a high-resolution hardware timer inside the SGX enclave necessitated the creation of software timers for defense, which unfortunately proved to be vulnerable. This scenario underlines a valuable lesson: hardware designers and software developers need to operate interdependently to bolster the security of the computing environment. Security threats like side-channel attacks are not exclusively the domain of either software developers or hardware designers. Instead, they demand a cooperative and collaborative approach to problem-solving, highlighting the inseparable interconnectedness of software and hardware considerations when addressing complex security challenges.

For a summary of the above discussions, we emphasize the importance of hardware and software co-design for operating system security, which is each chapter of the thesis has in common. By integrating both hardware and software elements, it allows for a robust and resilient security system that leverages the strengths of each to bolster the other. This theme has been consistently echoed throughout this thesis, highlighting the necessity of adaptability and innovative thinking in facing the dynamic landscape of operating system security threats. By examining each chapter's focus within the scope of software and hardware combined design strategy, we gain a comprehensive understanding of its value in achieving efficient, effective, and cost-sensitive security solutions.

## 6.2 Conclusion and Future Work

In this thesis, we have explored the challenges associated with operating system security in the context of implementing hardware security extensions. The complexity of software security properties often contrasts with the simplified security models employed by hardware extensions. This discrepancy can lead to situations where software developers must deviate from the original design intent of the hardware extensions, whether for benign or malicious purposes.

We have examined three cases in this thesis, each focusing on distinct security issues, such as memory protection, TEE OS security applications, and side-channel attacks and defense within TEEs. The common lessons gleaned from these research projects can be summarized as follows:

The standard usage of hardware security extensions, as proposed by CPU designers, may not always fully align with the ideal expectations of software security scenarios and requirements. For instance, in the case of LMP, the original use of MPX instructions can impose performance penalties in certain situations, such as when using second-level tables for bound storage. Similarly, ARM TrustZone was not initially designed to support third-party applications, and some hardware resources, like monotonic counters, are limited due to wear-out concerns. In these cases, software system developers must devise innovative ways to adapt the standard use of the hardware to address security requirements and usage scenarios.

Hardware security extension designs may not initially account for a comprehensive range of security threats. For example, in the case of the SGX side-channel problem, cache channels were not considered the responsibility of the CPU design. As a result, software developers must leverage available hardware resources to create defenses that must be rigorously verified to ensure that they uphold the desired security properties.

In conclusion, the relationship between hardware security extensions and software security properties is a complex and evolving landscape. This thesis highlights the need for ongoing collaboration between hardware designers and software developers to create robust, adaptable, and secure systems that can effectively address the ever-changing threat landscape.

In our future work, we aim to broaden the scope of operating system security research by exploring new hardware extensions, such as Intel SGX 2.0, AMD SEV (Secure Encrypted Virtualization), and ARM CMSE (Cortex-M Security Extensions). These cutting-edge technologies present unique opportunities and challenges for securing software systems, and understanding their potential impact on operating system security is crucial. Furthermore, the Aion attacks research has inspired us to investigate defense mechanisms that can effectively detect and prevent such attacks while minimizing performance degradation. Balancing security and performance is a critical aspect of designing robust solutions, and our future work will focus on achieving this delicate balance. By extending our research to encompass these new hardware extensions and novel defense strategies, we hope to contribute to the ongoing efforts to strengthen the security and resilience of software systems in the face of evolving threats and emerging technologies.

# Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-flow Integrity”. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security.* Alexandria, Virginia, Nov. 2005.

[2] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. “Cache based remote timing attack on the AES”. In: *Proceedings of the 7th Cryptographers' track at the RSA conference on Topics in Cryptology (CT-RSA'07).* San Francisco, CA, Feb. 2007.

[3] AliPay. *Open platform of AliPay (translated).* https://doc.open.alipay.com/. Last accessed: 2023-06-02. 2017.

[4] Starr Andersen and Vincent Abella. *Data Execution Prevention.* https://technet.microsoft.com/en-us/library/bb457155.aspx. 2004.

[5] Apple. *iOS Security.* https://www.apple.com/ca/business/site/docs/iOS_Security_Guide.pdf. Last accessed: 2023-06-02. Jan. 2018.

[6] ARM. *ARM Generic Interrupt Controller Architecture Specification.* https://static.docs.arm.com/ihi0069/d/IHI0069D_gic_architecture_specification.pdf. Last accessed: 2023-06-02. 2019.

[7] ARM. *ARM Trusted Firmware - version 1.2.* https://github.com/ARM-software/arm-trusted-firmware/tree/77b05323921c23e4261ddd8fee5c326a79b0af97. Last accessed: 2023-06-02. 2017.

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. “SCONE: Secure Linux Containers with Intel SGX”. In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation.* Savannah, GA, USA, Nov. 2016.

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding Applications from an Untrusted Cloud with Haven”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation.* Broomfield, CO, USA, Oct. 2014.

[10] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. “Sliding right into disaster: Left-to-right sliding windows leak”. In: *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES).* Taipei, Taiwan, Sept. 2017.

[11] Alessandro Bertani, Marco Bonelli, Lorenzo Binosi, Michele Carminati, Stefano Zanero, and Mario Polino. “Untangle: Aiding Global Function Pointer Hijacking for Post-CET Binary Exploitation”. In: *Proceedings of the 20th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA ’23)*. Hamburg, Germany, July 2023.

[12] Jiang Bian, Remzi Seker, and Umit Topaloglu. “Off-the-record instant messaging for group conversation”. In: *Proceedings of the IEEE International Conference on Information Reuse and Integration*. Las Vegas, Nevada, Aug. 2007.

[13] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. “Protecting C++ Dynamic Dispatch Through VTable Interleaving”. In: *Proceedings of the 23rd Annual Networked & Distributed System Security Symposium (NDSS)*. San Diego, California, 2016.

[14] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *Proceedings of the USENIX WOOT*. Vancouver, Canada, 2017.

[15] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. “CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2019.

[16] Billy Bob Brumley and Risto M Hakala. “Cache-timing template attacks”. In: *Proceedings of the 15th Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2009)*. Tokyo, Japan, Dec. 2009.

[17] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas Wenisch, Yuval Yarom, and Raoul Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *Proceedings of the 27th USENIX Security Symposium*. Baltimore, USA, Aug. 2018.

[18] 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: *Proceedings of USENIX Security*. Vancouver, Canada, 2017.

[19] Miguel Castro, Manuel Costa, and Tim Harris. “Securing Software by Enforcing Data-flow Integrity”. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, Nov. 2006.

[20] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. “Return-oriented Programming Without Returns”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. Chicago, IL, 2010.

[21] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. “SGXPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution”. In: *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. Stockholm, Sweden, 2019.

[22] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. “Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu”. In: *Proceedings of the 2017 on Asia Conference on Computer and Communications Security (ASIACCS’17)*. Abu Dhabi, UAE, 2017.

[23] 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: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ASPLOS'08. Seattle, WA, USA, 2008.

[24] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert Deng. "ROPecker: A Generic and Practical Approach for Defending against ROP Attacks". In: *Proceedings of the 21st Annual Networked & Distributed System Security Symposium (NDSS)*. San Diego, California, 2014.

[25] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. "Real time detection of cache-based side-channel attacks using hardware performance counters". In: *Journal of Applied Soft Computing* volume 49 (Dec. 2016), pp. 1162–1174.

[26] Tzi-Cker Chiueh and Fu-Hau Hsu. "RAD: A Compile-Time Solution to Buffer Overflow Attacks". In: *Proceedings of the The 21st International Conference on Distributed Computing Systems*. Washington, DC, 2001.

[27] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. "HCFI: Hardware-enforced Control-Flow Integrity". In: *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*. New Orleans, Louisiana, Dec. 2016.

[28] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. "The last mile: An empirical study of timing channels on seL4". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Scottsdale, AZ, Nov. 2014, pp. 570–581.

[29] Matthew Cole and Aravind Prakash. "Simplex: Repurposing Intel Memory Protection Extensions for Secure Storage". In: *Proceedings of the Secure IT Systems: 27th Nordic Conference*. Reykjavic, Iceland, Nov. 2022.

[30] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. "Protecting Data on Smartphones and Tablets from Memory Attacks". In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Istanbul, Turkey, Mar. 2015.

[31] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. URL: https://ia.cr/2016/086. 2016.

[32] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. "Thwarting cache side-channel attacks through dynamic software diversity." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS '15)*. San Diego, CA, Feb. 2015, pp. 8–11.

[33] John Criswell, Nathan Dautenhahn, and Vikram Adve. "Virtual Ghost: Protecting Applications from Hostile Operating Systems". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, Utah, 2014.

[34] Cyrus Lee. *Huawei Pay launched in China*. http://www.zdnet.com/article/huawei-pay-launched-in-china/. Last accessed: 2023-06-02. 2016.

[35] Fergus Dall, Gabrielle Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks". In: *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Amsterdam, Netherlands, 2018.

[36] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. "The Performance Cost of Shadow Stacks and Stack Canaries". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. Singapore, 2015.

[37] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. "HAFIX: Hardware-assisted Flow Integrity Extension". In: *Proceedings of the 52nd Annual Design Automation Conference*. San Francisco, California, 2015.

[38] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". In: *Proceedings of the 35th IEEE Symposium on Security and Privacy*. San Jose, California, May 2014.

[39] Asmit De, Aditya Basu, Swaroop Ghosh, and Trent Jaeger. "Hardware assisted buffer protection mechanisms for embedded RISC-V". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* volume 39.issue 12 (2020), pp. 4453–4465.

[40] Remi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. "Camouflage: Hardware-assisted CFI for the ARM Linux kernel". In: *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*. San Fransico, CA, July 2020.

[41] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX". In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security '17)*. Vancouver, BC, Canada, Aug. 2017, pp. 51–67. ISBN: 978-1-931971-40-9.

[42] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks". In: *ACM Transactions on Architecture and Code Optimization (TACO)* volume 8.issue 4 (2012), pp. 1–21.

[43] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. "Cacheaudit: A tool for the static analysis of cache side channels". In: *ACM Transactions on Information and System Security (TISSEC)* volume 18.issue 1 (2015), pp. 1–32.

[44] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. "The Matter of Heartbleed". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. Vancouver, BC, Canada, 2014.

[45] Serge Egelman, Lorrie Cranor, and Jason Hong. "You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings". In: *Proceedings of Computer Human Interaction 2008*. Florence, Italy, May 2008.

[46] Jan-Erik Ekberg. “Securing Software Architectures for Trusted Processor Environments”. 2013. PhD thesis. Department of Computer Science and Engineering: Aalto University, May 2013.

[47] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. “XFI: Software Guards for System Address Spaces”. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Seattle, WA, Nov. 2006.

[48] Mathieu Escouteloup, Ronan Lashermes, Jean-Louis Lanet, and Jacques Jean-Alain Fournier. “Recommendations for a radically secure ISA”. In: *Proceedings of the CARRV Workshop on Computer Architecture Research with RISC-V*. Online, May 2020.

[49] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. “Missing the Point(er): On the Effectiveness of Code Pointer Integrity”. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy*. San Jose, California, May 2015.

[50] Zhu Feng and Iansiti Marco. “Entry into platform-based markets”. In: *Strategic Management Journal* volume 33.issue 1 (June 2011), pp. 88–106.

[51] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. “Komodo: Using Verification to Disentangl Secure-enclave Hardware from Software”. In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, Oct. 2017.

[52] Ivan Fratrić. *ROPGuard: Runtime Prevention of Return-Oriented Programming Attacks*. http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf. 2012.

[53] Kittur Ganesh. *Pointer Checker: Easily Catch Out-of-Bounds Memory Accesses*. https://software.intel.com/sites/products/parallelmag/singlearticles/issue11/7080_2_IN_ParallelMag_Issue11_Pointer_Checker.pdf.

[54] Xinyang Ge, Weidong Cui, and Trent Jaeger. “Griffin: Guarding control flows using Intel processor trace”. In: *ACM SIGPLAN Notices* volume 52.issue 4 (2017), pp. 585–598.

[55] Daniel Genkin, Luke Valenta, and Yuval Yarom. “May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS ’17)*. Dallas, TX, Oct. 2017.

[56] GlobalPlatform. *Trusted Execution Environment (TEE) Device Specifications*. https://www.globalplatform.org/specificationsdevice.asp. Last accessed: 2023-06-02. 2017.

[57] Michael Godfrey and Mohammad Zulkernine. “A server-side solution to cache-based side-channel attacks in the cloud”. In: *Proceedings of the 2013 IEEE 6th International Conference on Cloud Computing*. Santa Clara, CA, June 2013.

[58] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. “Out Of Control: Overcoming control-flow integrity”. In: *Proceedings of the 35th IEEE Symposium on Security and Privacy*. San Jose, California, May 2014.

[59] Javiser Gonzalez. *Generic TrustZone Driver in Linux Kernel*. https://lwn.net/Articles/623380/. Last accessed: 2023-06-02. 2014.

[60] Google. *Trusty TEE.* https://source.android.com/security/trusty/. Last accessed: 2023-06-02. 2019.

[61] Johannes Gözfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. “Cache Attacks on Intel SGX”. In: *Proceedings of the European Workshop on System Security (EuroSec).* Belgrade, Serbia, July 2017.

[62] 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: *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17).* Vancouver, BC, Canada, Aug. 2017.

[63] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: a fast and stealthy cache attack”. In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '16).* Donostia-San Sebastián, Spain, July 2016.

[64] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache template attacks: Automating attacks on inclusive last-level caches”. In: *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15).* Washington, D.C., Aug. 2015.

[65] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. “PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace”. In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY '17).* Scottsdale, Arizona, Mar. 2017.

[66] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. “Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory”. In: *Proceedings of IEEE Symposium on Security and Privacy.* San Jose, USA, 2015.

[67] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. “TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone”. In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services.* Niagara Falls, NY, USA, June 2017.

[68] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache game – Bringing access-based cache attacks on AES to practice”. In: *Proceedings of the IEEE Symposium on Security and Privacy.* Oakland, CA, May 2011.

[69] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. “Comparing Cache Architectures and Coherency Protocols on X86-64 Multicore SMP Systems”. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* New York, NY, USA, Dec. 2009.

[70] Marcus Hähnel, Weidong Cui, and Marcus Peinado. “High-Resolution Side Channels for Untrusted Operating Systems”. In: *Proceedings of USENIX Annual Technical Conference (ATC).* Santa Clara, USA, 2017.

[71] Daniel Hein, Johannes Winter, and Andreas Fitzek. *Secure Block Device Library.* http://www.iaik.tugraz.at/content/research/opensource/secure_block_device/. Last accessed: 2023-06-02. 2019.

[72] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. "InkTag: Secure Applications on an Untrusted Operating System". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. Houston, Texas, Mar. 2013.

[73] Hong Hu, Shweta Shinde, Adrian Sendroiu, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks". In: *Proceedings of the 37th IEEE Symposium on Security and Privacy*. San Jose, California, May 2016.

[74] Wei Huang, Zhen Huang, and David Lie. "LMP: Light-Weighted Memory Protection with Hardware Assistance". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*. Dec. 2016. URL: https://security.csl.toronto.edu/wp-content/uploads/2018/06/whuang_lmp_acsac2016.pdf.

[75] Wei Huang, Vasily Rudchenko, He Shuang, Zhen Huang, and David Lie. *Mint-TEE: Minimum TrustZone OS Supporting Secure Application Functionalities*. Oct. 2019. URL: https://weihuang.info/papers/mint-tee.pdf.

[76] Wei Huang, Vasily Rudchenko, He Shuang, Zhen Huang, and David Lie. "Pearl-TEE: supporting untrusted applications in TrustZone". In: *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX 2018)*. Toronto, ON, Canada, Oct. 2018.

[77] Wei Huang, Shengjie Xu, Yueqiang Cheng, and David Lie. "Aion Attacks: Manipulating Software Timers in Trusted Execution Environment". In: *18th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. July 2021. URL: https://security.csl.toronto.edu/wp-content/uploads/2021/06/whuang-dimva2021-aion_v2.pdf.

[78] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks Against Kernel Space ASLR". In: *Proceedings of the 34th IEEE Symposium on Security and Privacy*. Washington, D.C., 2013.

[79] IFAA. *Internet Finance Authentication Alliance*. https://ifaa.org.cn/en. Last accessed: 2023-06-02. 2017.

[80] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cache attacks enable bulk key recovery on the cloud". In: *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Santa Barbara, CA, Aug. 2016, pp. 368–388.

[81] Intel. *Control-flow Enforcement Technology Preview, Document Number: 334525-001, Revision 1.0.* https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. June 2016.

[82] Intel. *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations*. June 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html.

[83] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture. Chapter 16: Programming with Intel Transactional Synchronization Extensions*. Intel Corporation. 2023.

[84] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3: System Programming Guide. Chapter 34: Introduction to Intel Software Guard Extensions*. Intel Corporation. 2023.

[85] Intel. *Intel Software Guard Extensions SDK*. https://software.intel.com/en-us/sgx-sdk-dev-reference-sgx-create-monotonic
-counter. Last accessed: 2023-06-02. 2018.

[86] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Lucky 13 strikes back". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS '15)*. Singapore, Apr. 2015.

[87] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES". In: *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2014)*. Gothenburg, Sweden, Sept. 2014.

[88] Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment". In: *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*. San Diego, CA, USA, Feb. 2015.

[89] Simon Johnson. *Intel SGX and Side-Channels*. Intel Corporation. 2017. URL: https://software.intel.com/en-us/articles/intel-sgx-and-side-channels.

[90] Uri Kanonov and Avishai Wool. "Secure Containers in Android: The Samsung KNOX Case Study". In: *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. Vienna, Austria, Oct. 2016.

[91] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N Serpanos, and Stefanos Kaxiras. "Non-deterministic caches: A simple and effective defense against side channel attacks". In: *Design Automation for Embedded Systems* volume 12.issue 3 (2008), pp. 221–230.

[92] Sharjeel Khan, Girish Mururu, and Santosh Pande. "Biscuit: A compiler assisted scheduler for detecting and mitigating cache-based side channel attacks". In: *arXiv preprint arXiv:2003.03850* (2020).

[93] Chongkyung Kil, Jinsuk Jim, C. Bookholt, J. Xu, and Peng Ning. "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software". In: *Proceedings of Computer Security Applications Conference (ASAC)*. Miami Beach, Florida, Dec. 2006.

[94] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. "Stealthmem: System-level protection against cache-based side channel attacks in the cloud". In: *Proceedings of the 21st USENIX Security Symposium (Security '12)*. Bellevue, WA, Aug. 2012.

[95] Alexander Klimov and Adi Shamir. "A New Class of Invertible Mappings". In: *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. San Francisco Bay, CA, USA, Aug. 2002.

[96] 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: *arXiv:1801.01203 [cs]* (Jan. 3, 2018). arXiv: 1801.01203. URL: http://arxiv.org/abs/1801.01203.

[97] Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. "Deconstructing new cache designs for thwarting software cache-based side channel attacks". In: *Proceedings of the 2nd ACM Workshop on Computer Security Architectures (CSAW '08)*. Alexandria, VA, Oct. 2008.

[98] Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. "Hardware-software integrated approaches to defend against software cache-based side channel attacks". In: *Proceedings of the 15th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Raleigh, NC, Feb. 2009.

[99] Gnanambikai Krishnakumar and Chester Rebeiro. "MSMPX: Microarchitectural Extensions for Meltdown Safe Memory Protection". In: *Proceedings of the 32nd IEEE International System-on-Chip Conference (SOCC)*. Singapore, Sept. 2019.

[100] Yusuf Kulah, Berkay Dincer, Cemal Yilmaz, and Erkay Savas. "SpyDetector: An approach for detecting side-channel attacks at runtime". In: *International Journal of Information Security* volume 18.issue 4 (2019), pp. 393–422.

[101] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. Broomfield, Colorado, Oct. 2014.

[102] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. "AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone". In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. Florence, Italy, May 2015.

[103] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. "Building trusted path on untrusted device drivers for mobile devices". In: *Proceedings of 5th Asia-Pacific Workshop on Systems*. Beijing, China, June 2014.

[104] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. "Sapper: A language for hardware-level security policy enforcement". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Salt Lake City, UT, Mar. 2014.

[105] Linaro. *OP-TEE*. https://www.op-tee.org/. Last accessed: 2023-06-02. 2018.

[106] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. "Take a way: Exploring the security implications of AMD's cache way predictors". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS '20)*. Taipei, Taiwan, Oct. 2020.

[107] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. "CATalyst: Defeating last-level cache side channel attacks in cloud computing". In: *Proceedings of the 22nd IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*. Barcelona, Spain, Mar. 2016.

[108] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B Lee. "Newcache: Secure cache architecture thwarting cache side-channel attacks". In: *IEEE Micro* volume 36.issue 5 (2016), pp. 8–16.

[109] Fangfei Liu, Yuval Yarom, Qian. Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *Proceedings of IEEE Symposium on Security and Privacy*. San Jose, USA, 2015.

[110] Limin Liu, Jin Han, Debin Gao, Jiwu Jing, and Daren Zha. “Launching Return-Oriented Programming Attacks Against Randomized Relocatable Executables”. In: *Proceedings of the 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. Changsha, China, Nov. 2011.

[111] Tingting Lu and Junfeng Wang. “Data-flow bending: On the effectiveness of data-flow integrity”. In: *Computers & Security* volume 84.issue C (2019), pp. 365–375. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2019.04.002.

[112] Tarjei Mandt, Mathew Solnik, and David Wang. “Demystifying the Secure Enclave Processor”. In: *Blackhat US 2016*. Las Vegas, NV, USA, 2016. URL: https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf.

[113] André Martin, Cong Lian, Franz Gregor, Robert Krahn, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. “ADAM-CS: Advanced Asynchronous Monotonic Counter Service”. In: *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Taipei, Taiwan, June 2021.

[114] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. “ROTE: Rollback Protection for Trusted Execution”. In: *Proceedings of the 26th USENIX Security Symposium (Security 17)*. Vancouver, BC, Canada, Aug. 2017.

[115] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters”. In: *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2015)*. Kyoto, Japan, Nov. 2015.

[116] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Mar. 2017.

[117] Microchip. *ATECC508A Crypto Authentication Device Complete Data Sheet*. http://ww1.microchip.com/downloads/en/DeviceDoc/20005927A.pdf. Last accessed: 2023-06-02. 2019.

[118] Richard Ta-Min, Lionel Litty, and David Lie. “Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable”. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, USA, Nov. 2006.

[119] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “CacheZoom: How SGX Amplifies The Power of Cache Attacks”. In: *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Taipei, Taiwan, 2017.

[120] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. “CopyCat: Controlled instruction-level attacks on enclaves”. In: *Proceedings of the 29th USENIX Conference on Security Symposium (Security ’20)*. Online, Aug. 2020.

[121] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. "Opaque Control-Flow Integrity". In: *Proceedings of the 22nd Annual Networked & Distributed System Security Symposium (NDSS)*. San Diego, California, 2015.

[122] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. "Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters". In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '18)*. Los Angeles, CA, June 2018.

[123] Hadi Nahari. "TLK: A FOSS Stack for Secure Hardware Tokens". In: *W3C Workshop on Authentication, Hardware Tokens and Beyond*. Mountain View, CA, USA, Sept. 2014.

[124] Michael Neve and Jean-Pierre Seifert. "Advances on Access-driven Cache Attacks on AES". In: *Proceedings of Selected Areas in Cryptography workshop (SAC)*. Berlin, Heidelberg, Aug. 2006.

[125] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. "TrustZone Explained: Architectural Features and Use Cases". In: *Proceedings of the IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. Pittsburgh, PA, Nov. 2016.

[126] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers". In: *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Atlanta, GA, Sept. 2017.

[127] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Intel MPX explained: A cross-layer analysis of the Intel MPX system stack". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)* volume 2.issue 2 (2018), pp. 1–30.

[128] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Andre Martin, Christof Fetzer, and Mark Silberstein. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. Boston, USA, July 2018.

[129] OpenSSL. *OpenSSL 1.1.1-dev*. https://github.com/openssl/openssl. Last accessed: 2023-06-02. 2017.

[130] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and countermeasures: the case of AES". In: *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'06)*. San Jose, CA, Feb. 2006.

[131] P. Saint-Andre. *End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)*. https://tools.ietf.org/html/rfc3923. Last accessed: 2023-06-02. Oct. 2004.

[132] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing". In: *Proceedings of the 22nd USENIX Security Symposium (Security '13)*. Washington, D.C., Aug. 2013.

[133] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. “Memoir: Practical State Continuity for Protected Modules”. In: *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. Oakland, CA, May 2011.

[134] PaX-Team. *PaX ASLR (Address Space Layout Randomization)*. http://pax.grsecurity.net/docs/aslr.txt. 2003.

[135] PayPal. *Mobile Payment Libraries Getting Started Guide*. https://developer.paypal.com/docs/classic/mobile/gs_MPL/. Last accessed: 2023-06-02. 2017.

[136] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V. Madhyastha, and Cristian Ungureanu. “Simba: Tunable End-to-end Data Consistency for Mobile Apps”. In: *Proceedings of the Tenth European Conference on Computer Systems (EuroSys ’15)*. Bordeaux, France, Apr. 2015.

[137] Nicole Perlroth. *Software Meant to Fight Crime Is Used to Spy on Dissidents*. The New York Times: https://www.nytimes.com/2012/08/31/technology/finspy-software-is-tracking-political
-dissidents.html. Last accessed: 2023-06-02. Aug. 2012.

[138] J. Pincus and B. Baker. “Beyond stack smashing: recent advances in exploiting buffer overruns”. In: *IEEE Journal of Security and Privacy* volume 2.issue 4 (July 2004), pp. 20–27.

[139] Sandro Pinto and Nuno Santos. “Demystifying ARM TrustZone: A Comprehensive Survey”. In: *ACM Computing Survey* volume 51.issue 6 (Jan. 2019). ISSN: 0360-0300. URL: https://doi.org/10.1145/3291047.

[140] Martin Pirker and Daniel Slamanig. “A Framework for Privacy-Preserving Mobile Payment on Security Enhanced ARM TrustZone Platforms”. In: *Proceedings of the 11th International Conference on Trust, Security and Privacy in Computing and Communications*. Liverpool, UK, June 2012.

[141] Moinuddin K. Qureshi. “CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping”. In: *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka, Japan, 2018.

[142] Moinuddin K. Qureshi. “New attacks and defense for encrypted-address cache”. In: *Proceedings of the 46th International Symposium on Computer Architecture (ISCA ’19)*. Phoenix, Arizona, June 2019.

[143] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. “fTPM: A Software-Only Implementation of a TPM Chip”. In: *Proceedings of the 25th USENIX Security Symposium*. Austin, TX, USA, Aug. 2016.

[144] Ashay Rane, Calvin Lin, and Mohit Tiwari. “Raccoon: Closing Digital Side-Channels through Obfuscated Execution”. In: *Proceedings of the 24th USENIX Security Symposium (Security ’15)*. Washington, D.C., Aug. 2015.

[145] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. “Return-Oriented Programming: Systems, Languages, and Applications”. In: *ACM Transaction on Information and System Security* volume 15.issue 1 (Mar. 2012), 2:1–2:34. ISSN: 1094-9224.

[146] Eyal Ronen, Kenneth G Paterson, and Adi Shamir. "Pseudo constant time implementations of TLS are only pseudo secure". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Toronto, ON, Canada, Oct. 2018.

[147] D. Rosenberg. "QSEE trustzone kernel integer over flow vulnerability". In: *Black Hat USA Conference*. Las Vegas, NV, USA, July 2014.

[148] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A Adam Ding. "SCADET: A side-channel attack detection tool for tracking Prime+Probe". In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD '18)*. San Diego, CA, Nov. 2018.

[149] Samsung. *Samsung Pay Security*. http://developer.samsung.com/tech-insights/pay/. Last accessed: 2023-06-02. 2017.

[150] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Salt Lake City, UT, Mar. 2014.

[151] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. "Virtual Monotonic Counters and Count-limited Objects Using a TPM Without a Trusted OS". In: *Proceedings of the First ACM Workshop on Scalable Trusted Computing (STC '06)*. Alexandria, VA, Nov. 2006.

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

[153] Jeff Seibert, Hamed Okhravi, and Eric Söderström. "Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. Scottsdale, Arizona, Nov. 2014.

[154] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the Effectiveness of Address-space Randomization". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. Washington, D.C., Oct. 2004.

[155] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring". In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSNW '11)*. Hong Kong, June 2011.

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

[157] Gal Shpantzer. *Implementing Hardware Roots of Trust: The Trusted Platform Module Comes of Age*. Tech. rep. Last accessed: 2023-06-02. SANS Whitepaper, June 2013. URL: https://trustedcomputinggroup.org/wp-content/uploads/SANS-Implementing-Hardware-Roots-of-Trust.pdf.

[158] SierraWare. *SierraTEE Trusted Execution Environment*. https://www.sierraware.com/open-source-ARM-TrustZone.html. Last accessed: 2023-06-02. 2018.

[159] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization". In: *Proceedings of the 34th IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2013.

[160] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy Campbell. "Scheduling, isolation, and cache allocation: A side-channel defense". In: *Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E '18)*. Orlando, FL, Apr. 2018.

[161] Raoul Strackx, Bart Jacobs, and Frank Piessens. "ICE: A Passive, High-speed, State-continuity Scheme". In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. New Orleans, LA, Dec. 2014.

[162] Raoul Strackx and Frank Piessens. "Ariadne: A Minimal Approach to State Continuity". In: *Proceedings of the 25th USENIX Conference on Security Symposium (Security '16)*. Austin, TX, Aug. 2016.

[163] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. "Breaking the Memory Secrecy Assumption". In: *Proceedings of the Second European Workshop on System Security (EuroSec '09)*. Nuremburg, Germany, Mar. 2009.

[164] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. "TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Denver, CO, USA, Oct. 2015.

[165] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. "The Evolution of Android Malware and Android Analysis Techniques". In: *ACM Computing Survey* volume 49.issue 4 (Jan. 2017), 76:1–76:41.

[166] Tencent. *WeChat Payment API Development Documents*. https://pay.weixin.qq.com/wechatpay_guide/help_docs.shtml. Last accessed: 2023-06-02. 2018.

[167] The Hacker News. *Over 420 Banking Apps Found On Google Play Store are Targeted By Android Trojan*. http://thehackernews.com/2017/04/android-banking-malware.html. Last accessed: 2023-06-02. 2017.

[168] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Ulfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM". In: *Proceedings of the 23rd USENIX Security Symposium*. San Diego, California, Aug. 2014.

[169] Top News. *800,000 mobile phones infected prostrate worm virus*. www.top-news.top/news-12136193.html. Last accessed: 2023-06-02. May 2016.

[170] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. "On the Expressiveness of Return-into-libc Attacks". In: *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*. Menlo Park, California, 2011.

[171] Trustonic. *Trustonic Secured Platforms*. https://www.trustonic.com/solutions/trustonic-secured-platforms-tsp/. Last accessed: 2023-06-02. 2018.

[172] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX’17)*. Shanghai, China, Oct. 2017.

[173] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. “Scheduler-based defenses against cross-VM side-channels”. In: *Proceedings of the 23rd USENIX Security Symposium (USENIX Security ’14)*. San Diego, CA, Aug. 2014.

[174] Pepe Vila, Boris Kopf, and Jose F. Morales. “Theory and Practice of Finding Eviction Sets”. In: *Proceedings of IEEE Symposium on Security and Privacy*. San Francisco, USA, May 2019.

[175] Sebastian Vogl, Robert Gawlik, Behrad Garmany, Thomas Kittel, Jonas Pfoh, Claudia Eckert, and Thorsten Holz. “Dynamic Hooks: Hiding Control Flow Changes Within Non-control Data”. In: *Proceedings of the 23rd USENIX Security Symposium (Security ’14)*. San Diego, CA, Aug. 2014.

[176] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. “Efficient Software-based Fault Isolation”. In: *SIGOPS Operating System Review* volume 27.issue 5 (Dec. 1993), pp. 203–216. ISSN: 0163-5980.

[177] Jingyuan Wang, Peidai Xie, Yongjun Wang, and Zelin Rong. “A Survey of Return-Oriented Programming Attack, Defense and Its Benign Use”. In: *Proceedings of the 13th Asia Joint Conference on Information Security (AsiaJCIS)*. Guilin, China, Aug. 2018.

[178] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. “Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation”. In: *Proceedings of the 28th USENIX Security Symposium (Security’19)*. Santa Clara, CA, Aug. 2019.

[179] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. “CacheD: Identifying Cache-Based Timing Channels in Production Software”. In: *Proceedings of the 26th USENIX Security Symposium (Security’17)*. Vancouver, BC, Canada, Aug. 2017.

[180] 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: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS’17)*. Dallas, USA, 2017.

[181] Zhenghong Wang and Ruby B Lee. “A novel cache architecture with enhanced performance and security”. In: *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Lake Como, Italy, Aug. 2008.

[182] Zhenghong Wang and Ruby B Lee. “New cache designs for thwarting software cache-based side channel attacks”. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA ’07)*. San Diego, CA, June 2007, pp. 494–505.

[183] Zhi Wang and Xuxian Jiang. “HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity”. In: *Proceedings of the 31st IEEE Symposium on Security and Privacy*. San Jose, CA, May 2010.

[184] Bryan C. Ward, Richard Skowyra, Chad Spensky, Jason Martin, and Hamed Okhravi. “The Leakage-Resilience Dilemma”. In: *Proceedings of the European Symposium on Research in Computer Security (ESORICS ’19)*. Luxembourg, Sept. 2019.

[185] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. "Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in ECDSA Implementations". In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. Online, Aug. 2020.

[186] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. "DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries". In: *Proceedings of the 27th USENIX Security Symposium (Security '18)*. Baltimore, MD, Aug. 2018.

[187] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "Microwalk: A framework for finding side channels in binaries". In: *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. San Juan, PR, Dec. 2018.

[188] Widevine. *Widevine DRM*. http://www.widevine.com/wv_drm.html. Last accessed: 2023-06-02. 2022.

[189] Johannes Winter. "Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms". In: *Proceedings of the 3rd ACM workshop on Scalable trusted computing*. Fairfax, VA, USA, Oct. 2008.

[190] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. "Eliminating timing side-channel leaks using program repair". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Amsterdam, Netherlands, July 2018.

[191] Xabber. *Introducing Xabber*. https://www.xabber.com/. Last accessed: 2023-06-02. 2022.

[192] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. "Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Dallas, TX, Oct. 2017.

[193] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. "CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Los Angeles, CA, 2022.

[194] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. "WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches". In: *Proceedings of the 44th IEEE Symposium on Security and Privacy*. San Fransico, CA, May 2023.

[195] Shengjie Xu, Wei Huang, and David Lie. "In-Fat Pointer: Hardware-Assisted Tagged-Pointer Spatial Memory Safety Defense with Subobject Granularity Protection". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Online, Apr. 2021. URL: https://security.csl.toronto.edu/wp-content/uploads/2021/03/xu-ifp-asplos2021.pdf.

[196] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *Proceedings of IEEE Security and Privacy*. Washington, USA, May 2015.

[197] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. “Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA ’17).* Toronto, ON, Canada, June 2017, pp. 347–360.

[198] Yuval Yarom and Katrina Falkner. “Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *Proceedings of the USENIX Security Symposium.* San Diego, USA, 2014.

[199] Bennet S. Yee. “A Sanctuary for Mobile Agents”. In: *Secure Internet Programming: Security Issues for Mobile and Distributed Objects* (1999), pp. 261–273.

[200] Gisu Yeo, Yeryeong Kim, Suhyeon Song, and Donghyun Kwon. “Efficient CFI Enforcement for Embedded Systems Using ARM TrustZone-M”. In: *IEEE Access* volume 10.issue 1 (2022), pp. 132675–132684. DOI: 10.1109/ACCESS.2022.3230791.

[201] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. “In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication”. In: *Proceedings of the 31st USENIX Security Symposium (Security 22).* Boston, MA, Aug. 2022.

[202] Bin Zeng, Gang Tan, and Greg Morrisett. “Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing”. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS ’18).* Chicago, IL, Oct. 2011.

[203] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. “VTrust: Regaining Trust on Virtual Calls”. In: *Proceedings of the 23rd Annual Networked & Distributed System Security Symposium (NDSS).* San Diego, California, 2016.

[204] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. “A hardware design language for timing-sensitive information-flow security”. In: *ACM SIGPLAN Notices* volume 50.issue 4 (2015), pp. 503–516.

[205] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. “Cloudradar: A real-time side-channel attack detection system in clouds”. In: *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2016).* Paris, France, Sept. 2016.

[206] Yinqian Zhang and Michael K Reiter. “Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS ’13).* Berlin, Germany, Nov. 2013.

[207] Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, and Dan Meng. “TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms”. In: *2016 IEEE Symposium on Computers and Communication (ISCC).* Messina, Italy, June 2016.

[208] Xianyi Zheng, Lulu Yang, Gang Shi, and Dan Meng. “Secure Mobile Payment Employing Trusted Computing on TrustZone Enabled Platforms”. In: *Proceedings of 2016 IEEE Trustcom/BigDataSE/ISPA.* Tianjin, China, Aug. 2016.

[209] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. “A software approach to defeating side channels in last-level caches”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS’16).* Vienna, Austria, Oct. 2016.

[210] Zongwei Zhou, Miao Yu, and Virgil Gligor. "Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O". In: *Proceedings of IEEE Sympsium on Security and Privacy*. San Jose, CA, USA, May 2014.