# HARDENING AND ADAPTING TRUSTED EXECUTION ENVIRONMENTS FOR EMERGING PLATFORMS

A Dissertation
Presented to
The Academic Faculty

By

Fan Sang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Cybersecurity and Privacy
College of Computing

Georgia Institute of Technology

August 2024

# HARDENING AND ADAPTING TRUSTED EXECUTION ENVIRONMENTS FOR EMERGING PLATFORMS

Thesis committee:

Prof. Taesoo Kim (Advisor)
School of Cybersecurity and Privacy
*Georgia Institute of Technology*

Dr. Brendan D. Saltaformaggio
School of Cybersecurity and Privacy
*Georgia Institute of Technology*

Dr. Sukarno Mertoguno
School of Cybersecurity and Privacy
*Georgia Institute of Technology*

Dr. Xiaokuan Zhang
Department of Computer Science
*George Mason University*

Dr. Ashish Kundu
Head of Cybersecurity Research
*Cisco Research*

Date approved: July 22, 2024

Shoot for the moon. Even if you miss, you’ll land among the stars.

*Norman Vincent Peale*

For my father.

## ACKNOWLEDGMENTS

This Ph.D. thesis could not have been completed without the help of many people.

First and foremost, I would like to express my sincere gratitude and appreciation to my advisor, Prof. Taesoo Kim. He placed his trust and confidence in my potential to carry out meaningful research by graciously accepting to mentor me as his Ph.D. student, despite my initial lack of expertise in system and software security. He has also offered invaluable guidance and unwavering support, not only during my research journey in this area but also when I was in deep sorrow over my family loss. His innovative ideas and feedback have significantly influenced all of my research projects over the past six years. More importantly, through his advice, he has taught me how to conduct critical thinking—the most crucial skill for an independent researcher. I am forever grateful for the training Taesoo has provided and aspire to be as excellent an advisor to my future students as Taesoo has been to me.

I would like to express my special thanks to Prof. Xiaokuan Zhang for the fruitful discussions and the help when we collaborated on SENSE, as well as for his invaluable career advice. I have learned a lot from working with him. I am also deeply grateful to Dr. Yu Ding, who not only mentored me during my internship at ByteDance but, most importantly, is the person who generously introduced me to the world of cybersecurity when I was just a curious undergraduate student and motivated me to pursue Ph.D. in security. His guidance has been invaluable from that time to the present. I am sincerely thankful to Dr. Ashish Kundu for kindly hosting me at Cisco Research during my internship. Many parts of this thesis stem from our fruitful collaboration, and the success of these research projects would not have been possible without his trust, support, and valuable feedback. Additionally, I want to thank Dr. Scott Constable, Dr. Yuan Xiao, Dr. Michael Steiner, and Mona Vij from Intel Labs for our years of engaging discussions and productive collaborations on Confidential Computing. Furthermore, I would like to thank Prof. Sukarno Mertoguno and Prof. Brendan D. Saltaformaggio for dedicating their time to serve on my thesis committee.

Their insightful comments have greatly improved this thesis.

This thesis is the result of excellent teamwork. I have had the privilege of working with numerous talented and friendly colleagues in SSLab, SCP, and at various internship locations. They have not only assisted me with research challenges but have also made daily life enjoyable. Given this opportunity, I would like to thank: Dr. Jaehyuk Lee, Dr. Yizhuo Zhai, Hangqing Zhao, Sujin Park, Mansour Alharthi, Ammar Askar, Mingyu Guan, Yu-Fu Fu, Kevin Stevens, Mingyi Liu, Fabian Fleischer, Xiang Cheng, Chuhong Yuan, Gyejin Lee, Andrew Chin, Jalen Chuang, Dr. Xin Jin, Charalampos Katsis, Jiahao Sun, and former SSLab members, Dr. HyungSeok Han, Dr. Soyeon Park, Prof. Seulbae Kim, Kuilin Li, Prof. Hang Zhang, Dr. Wen Xu, Yechan Bae, Jungwon Lim, Yonghwi Jin, Chulwon Kang, Stephen Tong, Dr. Jinho Jung, Dr. Ren Ding, Prof. Insu Yun, Prof. Hyungjoon Koo, Prof. Daehee Jang, Prof. Sanidhya Kashyap, Prof. Meng Xu, Prof. Hong Hu, Dr. Ming-Wei Shih, and Dr. Sangho Lee.

I want to express my deepest gratitude to my parents in China, Jianming Sang and Jieying Wang, for their unconditional love, patience, and support during my time at Georgia Tech. May you rest in peace, Dad. Lastly, to myself: thank you for believing in yourself and maintaining your enthusiasm for research. Those grinding nights will become the most cherished treasures in your life, and remember that only you have the power to accomplish your goals.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# SUMMARY

The rise of cloud computing, IoT, and edge computing has led users to often give up data control to third-party providers, raising security concerns. Trusted Execution Environments (TEEs), initially developed for cloud computing, create secure processor areas to protect sensitive data. However, TEEs are not yet integrated into emerging platforms due to their recency and ongoing development. Despite this, increasing security expectations and new privacy regulations necessitate adapting TEEs for these platforms. This thesis focuses on hardening and adapting TEEs for emerging platforms.

To optimally defend software under TEEs against multiple side-channel attacks (SCAs) on an arbitrary target platform, this thesis presents PRIDWEN, a novel framework that dynamically synthesizes a secure TEE program that is optimally hardened against various SCAs simultaneously, while preventing any deployability, redundancy, or incompatibility problem.

From a fundamental perspective, to harden TEEs at the root cause of microarchitectural SCAs, this thesis presents SENSE, an architectural extension that allows TEE programs to subscribe to fine-grained microarchitectural events, thus improving the microarchitectural awareness of TEEs and enabling proactive defenses previously unfeasible.

To enable TEEs on emerging platforms, this thesis finally presents PORTAL, a secure and efficient device I/O interface for Arm Confidential Compute Architecture (CCA) on modern mobile Arm processors. PORTAL addresses challenges due to memory encryption in the architectural trend of an increasing number of integrated devices within Arm processors. By leveraging Arm CCA's memory isolation mechanism, PORTAL enforces hardware-level access control without memory encryption. PORTAL offers robust security guarantees while eliminating the overhead of memory encryption, maintaining the performance and energy requirement crucial for emerging mobile platforms.

# CHAPTER 1
# INTRODUCTION

## 1.1 Problem Statement

The rise of cloud computing, IoT, edge computing, and the innovative domain of emerging platforms has shifted the paradigm of data control, compelling users to cede their digital information to third-party providers and raising pressing concerns about data privacy and security. Conducting *confidential or private computing* in a shared computing environment (e.g., the public cloud) is challenging [1, 2]. Trusted Execution Environments (TEEs), initially proposed for cloud computing, offer a potent remedy by establishing secure enclaves in processors that serve as sanctuaries for sensitive data. While the transition of TEEs into IoT and edge computing has been met with success, addressing unique security challenges inherent to these distributed settings, their adaptation to the emergent and rapidly developing platforms is still pending.

In this thesis, we focus on the integration of TEEs for emerging platforms. The process of adapting TEEs to rapidly developing platforms is a multifaceted endeavor. Despite these hurdles, the escalating demand for enhanced security among users and the advent of stringent privacy regulations underscore the urgency for TEEs to be adapted for emerging platforms.

First, it is crucial to recognize that current TEEs are not impervious to security threats. Strengthening these TEEs is an essential precursor to their successful implementation within emerging platforms. The ever-growing complexity and ubiquity of modern computing systems [3] have opened a Pandora's box of security challenges. One such challenge, side-channel attacks (SCAs) [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14], has emerged as a threat to the confidentiality and integrity of sensitive information. These attacks exploit shared resources, such as cache memory, to covertly extract secret data from seemingly secure systems. Modern TEEs (*e.g.*, Intel Software Guard Extensions (SGX) [15] and Trust Domain Extensions (TDX) [16], AMD Secure Encrypted Virtualization (SEV) [17], Arm TrustZone [18]) aim to provide architectural protection against privileged attackers (*e.g.*, OS and hypervisor) and consider some SCAs to be addressable by software, e.g., using constant-time programming techniques [19, 20, 10, 21, 22, 23, 24, 25, 26, 8, 27, 28]. Nonetheless, multiple side channels can co-exist in a vulnerable program; protecting TEE programs from multiple SCAs is difficult, while a fundamental solution at the root cause of SCAs is still not available.

Second, while the reinforcement of confidentiality and integrity is a pressing need for emerging platforms, the distinct requirements for performance, especially for the prevalent mobile computing platforms (*i.e.*, Arm platforms), take precedence. Maintaining the minimal performance requirements for mobile platforms is non-negotiable; without it, users are likely to abandon a platform regardless of any security enhancements. Therefore, any adaptation of TEEs to mobile platforms must not only augment security but also uphold, if not enhance, the overall performance. The historical trend in the development of mobile Arm processors, has been characterized by an increasing integration of diverse co-processors, peripherals, and devices alongside CPU. This trend began with the integration of basic elements such as memory management units and has evolved to include a wide range of specialized components such as Graphic Processing Units (GPUs), Neural Processing Units (NPUs), reality processors (*e.g.*, Apple R1 chip [29]) and various communication modules [30, 31, 32, 33, 34, 35, 36]. As the evolution of mobile Arm processors continues, these advancements are supporting emerging computing platforms that demand low latency, high power efficiency, substantial computational power, and maximum space efficiency within a single chip, such as in mobile devices, virtual reality, autonomous vehicles, and edge computing applications.

Meanwhile, the security measurement on Arm processors has also leaped forward significantly. Modern advancements in Confidential Computing [37, 38, 39, 40] have introduced confidential Virtual Machines (VMs) [17, 16, 41] which protect both the application and the entire VM software stack, including the guest Operating System (OS). Arm recently announced its adoption of confidential VMs, called Arm CCA [42]. Due to the strong security guarantees of confidential VMs, devices are untrusted and cannot directly access the protected memory allocated for confidential VMs. Workarounds for Arm CCA facilitate device communication through an untrusted *shared memory* region, such as `virtio` [43] or bounce buffer [44]. To meet the security requirements of confidential VMs, data in transit is encrypted when in untrusted shared memory. Unfortunately, Arm CCA aims to provide robust security mechanisms, but struggles to keep up with the trend of mobile Arm System on Chips (SoCs) [45], hindering its wider adoption upon the upcoming release.

This thesis aims to fill the gap of adapting TEEs for emerging platforms, by first devising techniques to harden existing TEEs against SCAs and then providing secure and efficient device Input/Output (I/O) interface for modern mobile Arm processors that demand pressing performance. The rest of this chapter will 1) explain how to address the limitations of existing techniques in defending SCAs against TEEs; 2) propose a secure and efficient device I/O interface for Arm CCA to fit the performance requirements for emerging mobile Arm platforms; and 3) summarize the contribution in this thesis work: adapting modern TEEs for emerging platforms.

## 1.2 Research Objectives

### 1.2.1 Hardening TEEs against Multiple SCAs Simultaneously

To defend against individual SCAs, software- and/or hardware-based countermeasures have been proposed, such as cache flushing [46, 47] or eviction detection [48], page fault detection [49], and HT disabling [46, 47] or co-location detection [50, 51]. However, multiple side channels can co-exist in a vulnerable program; protecting TEE programs from multiple known SCAs is difficult, not to mention the existence of unknown ones. One way is collectively applying existing countermeasures against individual SCAs, but naïvely doing so fails due to the unawareness of diverse target execution platforms or co-existing mitigation techniques, which may make such countermeasures 1) *undeployable* due to different hardware settings, 2) *redundant* because of over-protection, and 3) *incompatible* due to conflicts among different mitigations. Another way is adopting a comprehensive countermeasure, i.e., oblivious execution [52, 53], that is effective to many SCAs except for speculative execution. However, even the state-of-the-art oblivious execution incurs average slowdown of 51$\times$ [52], largely downgrading the cost-effectiveness of cloud computing. A practical alternative, data-location (re-)randomization [54], incurs relatively small slowdown (8$\times$), but it is still heavy and does not cover control-flow leakage.

Under this research objective, we focus on SGX programs, while the research scope can be extended to other TEEs in the future. One conventional approach to solve such issues is to create a *bloated* application incorporating independently compiled object files for each architecture and runtime detection code, to selectively activate them according to different hardware configurations [55]. This approach, however, is not suitable for Intel SGX: First, checking hardware configurations is supported by system software outside the TCB; malicious system software can provide misinformation about hardware configurations to SGX applications. Second, the secure memory that enclaves share, Processor Reserved Memory (PRM), is limited [56]; bloated SGX applications can easily occupy a huge portion of it.

To practically protect SGX programs from various SCAs, we argue that the decision of the SCA mitigations to be applied should be delayed as close to the final execution as possible to best fit the target SGX platform as well as co-existing mitigation techniques.

### 1.2.2 Hardening TEEs with Microarchitectural Awareness

Although disabling resource sharing entirely at the hardware level (*e.g.*, strict cache partitioning) mitigates many SCAs in TEEs, it is not practical and detriments utility. Constant-time programming is also challenging to deploy at scale. Sub-optimal hardware resource isolation techniques [57, 58, 59, 60, 61] try to find a balance between performance and security by allowing restricted resource sharing and dynamically partitioning a hardware structure (*e.g.*, cache) among multiple security domains. However, hardware-based mitigations, designed with existing knowledge and fixed at hardware level once deployed, cannot be easily upgraded and adapted to defend future SCA strategies. Recent studies [62] even show that dynamic hardware resource partitioning still leaks information and accurately quantifying the leakage is hard.

To be more flexible as well as potentially catch the missed leakage, various detection mechanisms [48, 63, 64, 65, 66, 67, 51, 68, 49] for TEEs have been proposed. *Detection-based* mitigations rely on observing the impact of SCAs on the victim's performance (*e.g.*, excessive cache misses) and detect ongoing attacks when anomalous performance characteristics are identified. Unfortunately, recent studies demonstrate that attackers can effectively bypass detection by leaking smaller portions of secrets (*e.g.*, 1 bit) across multiple victim executions instead of inferring most secrets within a single execution [69]. Consequently, abnormal performance behaviors are amortized across multiple executions, preventing detection tools from differentiating between benign and potentially malicious executions. In fact, any detection tool relying on the victim's performance characteristics is likely to fail [69]. As long as information leakage persists, adversaries can compensate for the scarcity of information by adopting more computationally intensive strategies, as demonstrated by formal SCA models [70].

It is indeed an irony that in microarchitectural SCAs against TEEs, attackers have the freedom to collect various microarchitectural signals, including those from the kernel space, while victim programs running in TEEs are constrained in their ability to reliably gather SCA signals and lack runtime awareness of the microarchitectural context, since the OS cannot be trusted. More specifically, detection-based techniques under TEEs encounter a number of limitations: 1) *missing trusted data sources*, exacerbated in TEEs due to untrusted OS mediation [63, 64, 65, 66, 67]; 2) *low-quality data*, leading to imprecise attack detection and allowing stealthy attacks [63, 64, 65, 66, 67, 51, 68]; 3) *inflexibility*, as victims have limited contextual awareness and can only make coarse-grained decisions [63, 64, 65, 66, 67, 51, 68, 49]; and 4) *platform constraints*, where techniques rely on platform-specific features, limiting extensibility and applicability to other architectures or future generations [48, 51, 68, 49].

Inspired by the practice of security through transparency (in contrast with security through obscurity), we believe that opening direct access to microarchitectural events with caution for TEEs can help to enable more complete forms of SCA defense strategies as well as facilitate use cases beyond just preventing SCAs. As history has proven, providing more transparency to the public with careful mediation (*e.g.*, open-source projects such as the Linux kernel and Kerckhoff's principle in cryptographic systems [71]) can contribute to the overall robustness of the system. To build a timely, accurate, flexible, and trustworthy technique that may fundamentally thwart microarchitectural SCAs, we argue that it is necessary to turn a *side channel* that can only be crafted by skilled attackers into a high-fidelity *direct channel* only accessible to victims in TEEs at runtime. This approach not only provides an immediate, effective, and future-proof response to potential attacks but also challenges the very foundation of existing SCA strategies, which relies on the information gap between the attacker and the victim.

### 1.2.3 Making Modern TEEs Practical for Emerging Platforms

The scalability and performance of Arm CCA are increasingly challenged as the number of integrated devices within mobile Arm System on a Chip (SoC) grows. For instance, customized Arm processors in modern autonomous vehicles [31, 72, 73, 74, 75], featuring CPUs, GPUs, sensor fusion processors, and AI accelerators (*e.g.*, NPUs), exemplify the complexity an Arm SoC must handle. When these devices concurrently access encrypted memory shared by all devices and the CPU, significant performance degradation occurs. In addition, memory encryption naturally conflicts with existing optimizations such as deduplication [76]. Encryption makes identical data blocks appear different, leading to inefficient memory usage, especially in systems requiring rapid, frequent access to shared data. These drawbacks impact real-time data processing in resource-constrained platforms such as autonomous vehicles, where minor lags can compromise safety and efficiency.

Furthermore, point-to-point memory encryption in current secure device I/O approaches for Arm CCA [77, 78, 79, 80] ties the device to the entire lifespan of the confidential VM, preventing dynamic device assignment during runtime, a common scenario when integrated devices on mobile SoCs are shared by multiple tenants. Lastly, unlike desktop platforms, mobile Arm platforms operate under stringent power efficiency requirements (*e.g.*, battery). Repetitive memory encryption and decryption significantly increase energy consumption [81, 82, 83, 84, 85, 86, 87]. The growing number of integrated devices that require secure interactions exacerbates this problem when memory is shared in the mobile Arm SoCs.

We propose a bold approach: *achieving Arm CCA secure device I/O by sole isolation without memory encryption*. Historically, memory encryption is deployed in Confidential Computing because if any of the isolation techniques have been compromised, the data being accessed is still protected by cryptography. One particular case is to prevent physical attacks such as memory probing [88, 89, 90, 91] and cold boot attacks [92, 93, 94, 95], as isolation does not extend to the memory bus. In other words, memory encryption can be redundant if memory isolation is *strictly* and *always* enforced. Thanks to the *integrated memory* design adopted in common mobile Arm SoCs, we argue that encrypting the shared memory between the confidential VMs and the peripheral devices is unnecessary. On the one hand, external attackers cannot launch physical attacks by probing the memory bus, as the integrated memory design connects the memory directly to the CPU and devices within the Arm SoC [96, 97, 98], which is commonly equipped with tamper detection against unauthorized physical accesses. On the other hand, unlike traditional process-based Confidential Computing (CC), Arm CCA enforces strict hardware-based memory access control even for privileged software (*i.e.*, OS and VMM) so that privileged attackers cannot access protected memory regions [37, 42]. As a result, memory encryption is unnecessary if no other entities except designated confidential VMs and devices can read or write to the shared memory region. In return, it is possible to achieve plaintext secure device I/O, which enhances Arm CCA for mobile platforms by 1) boosting the performance for secure data transmission, 2) retaining scalability with the increasing trend of integrated devices, and 3) reducing burden on energy efficiency.

## 1.3 Thesis Contribution

This thesis focuses on hardening and adapting TEEs for emerging platforms and addressing the various challenges that come up. In summary, this thesis makes the following contributions to advance the research area:

**PRIDWEN [§3].** In response to §1.2.1, this thesis presents PRIDWEN, a framework that uses *load-time synthesis* to dynamically harden SGX programs by selectively applying different mitigation techniques according to the configurations on the target execution platform. While ensuring the security, PRIDWEN maintains the cost-effectiveness of cloud computing by minimizing the extra effort required for preparation on the tenant side, and the runtime overhead of program synthesis on the cloud side.

PRIDWEN has a universal loader that securely loads and hardens a given SGX program inside an enclave based on four components: 1) Prober that identifies the target platform's hardware and system configurations using SGX exception handling logic and remote attestation; 2) PassManager that manages and determines an optimal set of feasible instrumentation passes based on the identified configurations; 3) Synthesizer that hardens a given SGX program with the chosen instrumentation passes before loading it in the target enclave; and 4) Validator that verifies whether the final executable is hardened as expected, and provides a functionality for developers to remotely verify both the process of synthesis and the hardened binary before execution.

To make PRIDWEN 1) *platform-independent*, 2) *instrumentation-friendly*, and 3) *lightweight*, we develop a new instrumentation framework using Wasm [99, 100] as the Intermediate Representation (IR). The size of PRIDWEN in binary is only 1.26 MiB, which only adds a slim footprint to the enclave TCB. Existing Wasm runtimes for SGX [101, 102, 103, 104] only *interpret* Wasm binaries without any *instrumentation support*. Furthermore, unlike existing Wasm instrumentation frameworks for non-SGX programs [105, 106], PRIDWEN can comprehensively transform Wasm binaries at both Wasm IR and native code levels. PRIDWEN also supports multiple high-level languages; users only need to compile their SGX programs once with a Wasm compiler backend (*e.g.*, Emscripten [107]).

To showcase the capability and practicality of PRIDWEN, we integrate four mitigation passes into PRIDWEN: 1) T-SGX [49] to prevent a page-fault SCA with a hardware support; 2) Varys [51] to mitigate cache-timing, page-fault-, and HT-based attacks in a software-only manner; 3) QSpectre [108] to mitigate the Spectre attack; and 4) fine-grained Address Space Layout Randomization (ASLR) [109] as a general-purpose mitigation. PRIDWEN poses moderate performance overhead on top of the original mitigation techniques and retains faithfulness of execution semantics (§3.4). The average slowdown of hardened real-world applications (Lighttpd, libjpeg, and SQLite) was 2.1× with hardware-assisted non-redundant mitigation techniques and 3.6× with software-only mitigation techniques for outdated microcode (*i.e.*, no hardware-level mitigation), which closely conforms to the originally reported performance overhead of the selected countermeasures. Also, PRIDWEN faithfully compiled and ran all 73 programs from the official Wasm specification test suite [110]. Program synthesis completed within 0.5 s with a temporary usage of less than 25 MiB of enclave memory across tests.

**SENSE [§4].** In response to §1.2.2, this thesis presents SENSE, a paradigm-shifting hardware-software co-design solution that directly exposes events at microarchitectural level to userspace TEEs. SENSE is a general architectural extension that provides a reliable source of precise microarchitectural information and a flexible method for feeding microarchitectural events directly to TEEs. It achieves this by *directly notifying* userspace software in TEEs about microarchitectural events, allowing users to *proactively defend* themselves against SCAs using actions specified in the event handler, such as enforcing security invariants (*e.g.*, pinning secret-dependent cache entries). As SENSE inevitably becomes part of the attacker's arsenal, we also conduct an in-depth security analysis of the SENSE architecture, SENSE handlers, and potential attack surfaces, and demonstrate that

SENSE does not leak more information than a system without SENSE does.

While SENSE is designed to be a generic framework, in this thesis, we enable SENSE specifically for thwarting cache-based SCAs for TEEs, as they are the most widely researched SCAs against TEEs. We prototype SENSE on a cycle-accurate gem5-based [111] emulator (§4.4). Evaluation results show that SENSE can successfully defeat cache SCAs and incurs negligible overhead (1.2%) on reasonable TEE workloads under benign situations. SENSE has more use cases than detecting SCAs. For example, secure software that typically incur high performance overhead (*e.g.*, constant-time cryptographic algorithms) can be dynamically loaded (*i.e.*, dynamic switching) in TEEs by SENSE handlers upon sensitive events, ensuring the performance penalty is only paid when necessary. SENSE can also be utilized to audit the faithfulness of an untrusted OS by verifying whether the contracts between the OS and userspace (*e.g.*, cache coloring) are honored, owing to the rich microarchitectural information provided by SENSE.

SENSE serves as the first advocate for a transparent and trustworthy source of high-fidelity microarchitectural information dedicated to TEEs.

**PORTAL [§5].** To fulfill §1.2.3, this thesis presents PORTAL, a secure and efficient device I/O interface for Arm CCA on mobile Arm SoCs. PORTAL achieves secure I/O by strict memory isolation without memory encryption. In essence, PORTAL provides a plaintext memory region whose access control is strictly enforced by Arm CCA so that no parties except the designated Realm VMs and peripherals can access it. PORTAL leverages the capabilities of Arm CCA's Granule Protection Check (GPC) and System Memory Management Unit (SMMU) to enforce hardware-level memory isolation. GPC enforces strict access control on the host physical address space and is mandatory for any component that generates memory transactions under the CCA model. SMMU provides integrated devices with virtual to physical address translation and access controls when accessing the host memory. The two trusted access control components collaborate to enforce two-way exclusive memory access between designated Realm VMs and devices, achieving secure I/O by isolation without encryption.

To ensure the integrity of the isolation, PORTAL employs a specialized Realm VM to protect the configuration and management of the sensitive data structures (*e.g.*, translation tables) that establish the isolation. As PORTAL removes the memory encryption and adopts a plaintext-based approach, we also conduct an in-depth security analysis of PORTAL and make sure that PORTAL does not open new attack surfaces. We prototype PORTAL on two official platforms, the Arm Fixed Virtual Platforms (FVPs) emulator [112] for software-simulated Arm CCA features and the Orange Pi 5 Plus [113] with Arm Mali-G610 GPU. Evaluation results show that PORTAL-based secure I/O incurs a minimal one-time overhead of 9.8% with runtime device management and can achieve a significant performance gain (1.07$\times$-9.07$\times$ on selected benchmarks) on data-intensive applications compared to memory encryption solutions.

PORTAL tries to foster a broader adoption of Arm CCA on the most widely deployed mobile processor architecture upon its official release in the near future. PORTAL serves as the first plaintext secure I/O interface for Arm CCA to achieve performance, scalability, and power efficiency on demanding mobile Arm platforms.

# CHAPTER 2
# BACKGROUND

## 2.1 Intel SGX

Intel SGX is a hardware-based TEE that securely runs a userspace application in an untrusted remote environment, such as the public cloud. Through remote attestation [114], Intel SGX allows a user to load his/her signed program into a remote environment, while helping ensure that the program binary has never been altered. To help secure the code and data of SGX programs, Intel SGX provides an enclave to each program, which is a dedicated secure region of the main memory. The enclave is isolated from any other software including an OS. The code and data stored in the enclave are always encrypted by the Memory Encryption Engine (MEE), and decrypted only when they are loaded into a CPU package (i.e., the cache), to help prevent physical attacks such as a cold boot attack [92].

**Exceptions in SGX.** Conventionally, exceptions are delivered to system software such as OS for further investigation. However, Intel SGX cannot adopt traditional exception handling because system software is untrusted. Instead, Intel SGX defines two mechanisms, Asynchronous Enclave Exit (AEX) and Custom Exception Handler (CEH) [115, 116]. Whenever an exception is generated during an enclave execution, the CPU exits from the enclave asynchronously. During AEX, the original exception number and register context are stored into the State Save Area (SSA) inside the enclave and then overwritten by synthetic data. Further, SGX allows developers to define CEHs to process exceptions inside an enclave; These CEHs can retrieve the SSA to check the stored exception number (`GPRSGX.EXITINFO.VECTOR`) and registers (`GPRSGX.<registers>`), and update them (e.g., `GPRSGX.RIP`) to resume the execution.

**SGX side channels.** SCAs against SGX have been actively studied. Traditional cache SCAs work against Intel SGX with different configurations [19, 117, 118, 119, 28, 23]. Recent studies show that page access patterns [25, 120, 121, 122, 53], interrupt execution time [8, 9], branch prediction behaviors [123, 10], speculative execution [20, 21] and Intel's internal buffers [124, 125, 126] can all be used to infer sensitive information inside enclaves. In response, countermeasures that cope with the fundamental characteristics of the SCAs have been proposed. T-SGX [49] and Cloak [48] use TSX to accurately recognize page faults and cache evictions inside an enclave, respectively. Varys [51] and Déjà Vu [68] aim to detect frequent interrupts or AEXs. Also, since HT enables concurrent SCAs without interrupts, Varys [51] and HyperRace [50] try to prevent an SGX hyperthread from being co-located with other hyperthreads. Disabling HT and/or flushing the L1 cache are also necessary to mitigate recent speculative or transient SCAs [46, 47]. In addition, SGX-LAPD [127] leverages a huge page to degrade the accuracy of the page-level SCA. Lastly, oblivious code execution and data access techniques for Intel SGX [52, 53, 128, 129, 130, 131] have been proposed as a general countermeasure against SCAs, but they incur overly high performance overhead. The state-of-the-art ORAM-based system Klotski [132] improves the performance significantly. However, it only defeats controlled-channel attacks [53, 25]. In contrast, PRIDWEN focuses on universally hardening SGX applications by automatically and selectively applying multiple defenses against different SCAs together.

## 2.2 WebAssembly (Wasm)

The World Wide Web Consortium (W3C) proposes Wasm [99] as a platform-independent compilation target for various high-level languages (e.g., `C/C++` and `Rust`). A Wasm binary has language-like syntax and structure that are suitable for compilation and instrumentation. The basic executable unit of code in Wasm is a module that consists of multiple sections, where each section contains specific definitions of the module such as global variables, functions and a sequence of instructions of each function. Wasm instructions execute on a stack machine, and Wasm supports only the structured control flow such as `if-else` and `loop` without `goto` statements, enabling single-pass fast compilation.

**Memory safety in Wasm.** Wasm maintains a linear memory with a configurable size dedicated to all the memory accesses except for local and global variables. The linear memory is disjoint from other memory regions such as the `code` section and the call stack. As a result, given a buggy Wasm program (e.g., originating from a `C` program with a memory corruption bug), an attacker can only interfere with the data in the linear memory, but cannot tamper with its control flow.

**PRIDWEN and Wasm.** Because the Wasm binary is well-structured and friendly for efficient Just-In-Time (JIT) compilation, PRIDWEN adopts Wasm as IR for supporting load-time synthesis. PRIDWEN also benefits from Wasm's memory-safety feature, mitigating code-reuse attacks against SGX [133, 134]. Moreover, the minimal footprint of Wasm fits PRIDWEN's demand for a compact yet flexible instrumentation and compilation toolchain inside an enclave. Although existing compilers (e.g., LLVM) can fit into enclaves with extended size in new scalable CPUs [135], the TCB size will largely increase, and the migration effort would be significant as well.

## 2.3 Microarchitectural Events

Microarchitecture comprises hardware components that are not directly accessible to software, as they are typically abstracted by the Instruction Set Architecture (ISA) and function (*e.g.*, accelerate certain operations) transparently from the software layer. Microarchitectural events are generally related to instruction executions and memory operations. Instruction-related microarchitectural events include overall instruction fetch, issue, dispatch (execution) and retirement. Memory-related microarchitectural events include load and store operations on various memory components, such as CPU cache hit and miss, Translation Lookaside Buffer (TLB) hit and miss, and Page Table Walks.

However, while not *directly* accessible to the software layer, microarchitectural events can be inferred *indirectly* by carefully controlled software execution, hence, leaking information about software activities in unexpected ways. Furthermore, as all programs running on a machine share the same set of hardware components (*i.e.*, microarchitectural states), attackers who can perform controlled execution of one program can leverage the observed microarchitectural events to infer the behavior of other programs. This forms the theoretical basis of side-channel attacks (SCAs).

## 2.4 Cache-based Side-Channel Attacks (SCAs)

Among a diverse set of SCAs, cache-based SCAs [4, 136, 6, 137, 138, 7, 139, 140, 141, 142, 5, 143] present a risk to secure computing across a variety of platforms and architectures, including TEEs such as Intel SGX [19, 144, 26, 27] and ARM TrustZone [142, 145]. These attacks can disclose both fine-grained and coarse-grained private data and operations, including bypassing address space layout randomization (ASLR) [138, 141], deducing keystroke patterns [139, 140], leaking sensitive information from human genome indexing computations [19], and exposing RSA [5, 143] and AES decryption keys [6, 146].

Cache-based SCAs exploit the time difference between cache hits and misses to infer secrets by learning whether specific cachelines have been accessed by the victim program. Commonly used techniques include Prime+Probe [4, 136, 6, 5, 137, 138] (to monitor cache set access patterns), and Flush+Reload [7, 139, 140, 141] (to evict shared target cachelines) while other variations have been proposed as well.

**Detection-based defenses.** To counter the aforementioned attacks, researchers have proposed various detection-based countermeasures [69]. A detection-based solution aims to identify ongoing attacks by monitoring program performance characteristics, such as cache miss rates and number of interrupts, to determine suspicious processes [63, 64, 65, 66, 68, 49, 48, 67]. We cover detection-based techniques in depth in related work of SENSE (§4.6).

**Limitations.** Detection-based strategies [63, 64, 65, 66, 68, 49, 48, 67] are based on heuristics and face numerous challenges:

*1) Missing trusted data sources in TEEs.* Although the scarcity of data sources also applies to defending SCAs without TEEs, it is significantly exacerbated under TEEs. While direct information about microarchitectural events (*e.g.*, cache events) is available through native interfaces (*e.g.*, performance counters), the reliance on untrusted OS mediation (*e.g.*, by registering a signal handler) renders these information sources inapplicable under TEEs [63, 64, 65, 66, 67], and no alternative sources for direct microarchitectural information exist.

*2) Low quality of available data.* The statistical and noisy nature of performance characteristics (*e.g.*, number of page-faults or interrupts), along with existing microarchitectural information primarily intended for performance tuning and runtime profiling purposes (*e.g.*, the number of cache misses), often leads to delayed and imprecise detection of attacks. This limitation allows for more "stealth" attacks [120, 28, 137] and leaves victims vulnerable to continued exploitation [63, 64, 65, 66, 67, 51, 68].

*3) Inflexible.* Due to the lack of detailed microarchitectural information in the userspace, victims have limited contextual awareness about underlying microarchitectural events and can only make coarse-grained decisions based on impressions. Existing techniques either terminate or retry the workload until success, restricting the flexibility of actions that victims can take [63, 64, 65, 66, 67, 51, 68, 49].

*4) Platform specific.* In exchange for sacrificing the effectiveness of defending SCAs in TEEs, detection-based techniques gain practicality by avoiding hardware modifications and using existing platform-specific features (*e.g.*, Intel TSX [48, 51, 68, 49]) to collect abnormal performance characteristics and thwart SCAs for TEEs on the corresponding platforms (*e.g.*, Intel SGX [48, 51, 68, 49]). However, relying on platform-specific hardware features makes these techniques non-extensible and inapplicable to other existing architectures or future generations, not to mention the potential risk of deprecation (*e.g.*, TSX deprecation [147, 148]), limiting the generality of the solutions.

**Reflection:** It is ironic that TEEs, which are designed to shield a program from external inferences, are now blocking the program from using detection-based SCA countermeasures proactively. Even when the program inside a TEE has perfect knowledge on how it might be attacked (*i.e.*, what microarchitectural signals to watch for), the victim lacks trusted medium to gather these signals, not to mention the performance penalties it has to pay for even coarse-grained data. In contract, attackers, not bounded by TEEs, have the freedom to gather all kinds of microarchitectural signals even from the kernel space.

## 2.5 Arm Integrated Memory

Integrated memory is a configuration where the memory subsystem is embedded onto the processor's silicon. This design allows shared access between the CPU and peripherals like GPUs, NPUs, Digital Signal Processors (DSPs), and co-processors, optimizing space, power efficiency, and data access speeds.

Integrated memory is prevalent in Arm processors, especially in mobile and embedded systems where compact design and energy efficiency are crucial [30, 31, 32, 33, 34, 35, 36]. This architecture is standard in smartphones, tablets, and portable devices [33, 36], which dominate the mobile processor market and benefit from the space efficiency and power savings of integrated memory. In addition, integrated memory is increasingly used in automotive systems [31, 72, 73, 74, 75], extended reality headsets [30, 29], industrial machines [34], and edge computing devices [35, 32]. This trend shows the demand for efficient, responsive, and robust computing across industries, making integrated memory essential in modern Arm processors.

**Feasibility of physical memory attacks.** Considering physical attacks [88, 89, 90, 91, 92, 93, 94, 95] on Arm systems with integrated memory, several factors render these attacks impractical. The compact and integrated nature of Arm SoCs makes memory components physically inaccessible without specialized tools and risk of significant damage [96, 97, 98]. Advancements in packaging technology [149, 150] protect these components by making

**Table 2.1:** Accessibility of physical memory pages having different Physical Address Space (PAS) from different processor security states. NS stands for Non-Secure.

| Security State | NS PAS | Secure PAS | Realm PAS | Root PAS |
|---|---|---|---|---|
| NS | ✓ | × | × | × |
| Secure | ✓ | ✓ | × | × |
| Realm | ✓ | × | ✓ | × |
| Root | ✓ | ✓ | ✓ | ✓ |

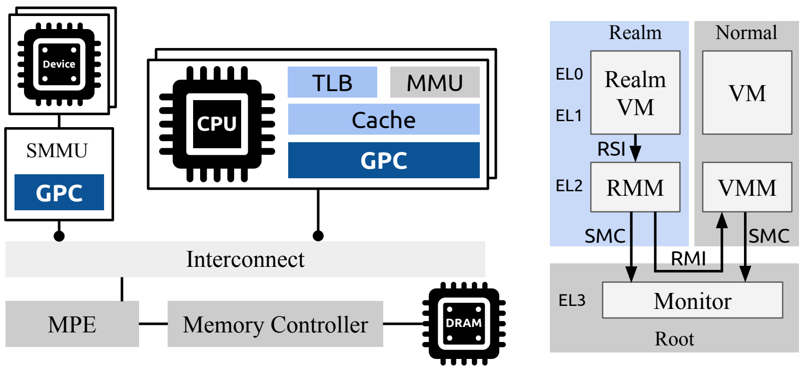

**Figure 2.1:** GPC in Arm CCA. Blue components are subject to GPC enforcement. The right figure shows interfaces between the Realm VM, RMM, VMM, and Monitor. The SMC instruction allows the RMM and VMM (EL2) to return control to the Monitor (EL3). Realm service interface (RSI) is the channel for requesting services from the RMM, and realm management interface (RMI) is the communication channel from the host to the RMM.

direct memory access exceedingly difficult without sophisticated equipment. Furthermore, security features such as tamper detection [151, 152, 153, 154, 155] safeguard against unauthorized physical access by triggering automatic data deletion or device locking upon tampering attempts. These protections greatly reduce the chances of successful physical attacks on integrated memory.

## 2.6 Arm CCA

Arm CCA [37, 42] enables the creation of VM-based trusted execution environments via a hardware extension to the Armv9 ISA [156], referred to as Realm Management Extensions (RME) [157]. RME introduces two new worlds, emphRealm and *Root*, in addition to the existing *Normal* and *Secure* worlds. CCA manages these worlds by implementing a GPT, a crucial data structure that assigns physical addresses (or Physical Address Spaces (PAS) in CCA terminology) to one of the four worlds, where a *granule* is the smallest unit of memory that can be managed. This setup enhances isolation by equipping Processing Elements (PE) (*i.e.*, Arm cores, SMMU, caches, and TLBs) with GPC, which check if the source and destination PAS conform to the access control policies (see Table 2.1). For example, when a CPU core executes in Realm mode, it can generate memory transactions for the Realm PAS. If an invalid transaction is detected, CCA triggers a Granule Protection Fault (GPF) and handles it in EL3.

The GPTs, stored in the main memory under the *Root* world, are accessible only by the trusted Monitor, which updates them as necessary. Changes to the GPTs trigger synchronization of the GPC, which requires flushing outdated states to maintain consistency. These checks are pivotal in preventing unauthorized access to the *Realm* memory from the untrusted *Normal* or *Secure* worlds by controlling all memory accesses and ensuring that memory transactions initiated from the core are securely managed.

The trusted Monitor at EL3 in the Root world adjusts the world bit during context switches, enhancing security. Concurrently, the trusted RMM at EL2 in the *Realm* world isolates confidential VMs by managing stage-2 translations from guest to host physical addresses. Lastly, memory encryption and integrity protection prevent physical tampering with main memory data.

## 2.7 Arm Peripherals and SMMU

Unlike discrete peripherals in Intel-based systems, Arm-based devices use a unified memory architecture (§2.5) shared with the CPU and other peripherals (*e.g.*, GPU and NPU). Arm handles data transfers and communication between the host and device through comprehensive software, including kernel drivers and user runtimes. This software manages the device computation environment and facilitates hardware interactions.

To set up the execution environment, the device software allocates physical memory and creates buffers for specific tasks. It then loads essential task elements into device memory. Additionally, the software stack creates the page table and configures registers for Direct

Memory Access (DMA) to access critical components. Interaction with hardware is managed via task scheduling and submission through Memory-Mapped Input/Output (MMIO). After computations, the software retrieves results and restores the system environment.

Given the shared memory architecture, Arm has integrated the SMMU to oversee DMA-capable peripherals. Most Arm GPUs [158, 159, 160] and other peripherals are connected to an SMMU. Similar to the CPU's Memory Management Unit (MMU), the SMMU [161] performs stage-1 and stage-2 address translations to regulate peripheral access to the physical address space. Privileged software configures the SMMU registers, including page table and translation configuration registers, through MMIO. Besides address translation, the SMMU supports GPC under Arm CCA. To secure these features, CCA adds SMMU MMIO registers accessible only in the *Root* world, offering configurations for SMMU GPC, such as GPT base, GPC controls, fault handling, and TLB invalidation.

**Device identity management.** Each device connected to the SMMU is assigned a unique *Stream ID (SID)* included in its memory access requests. The *Stream Table (ST)* maps SIDs to their corresponding *Context Descriptors (CDs)*, which contain the virtual-to-physical address translation table. To retrieve the physical address, the SMMU uses the device's SID to traverse the ST and CD for the translation table, which is then used to find the physical address. Besides address translation, the ST and CD in the SMMU manage memory access, access control, and security for connected devices.

# CHAPTER 3
# PRIDWEN: UNIVERSALLY HARDENING SGX PROGRAMS VIA LOAD-TIME SYNTHESIS

**Disclaimer:** *The two lead authors, Fan Sang and Ming-Wei Shih, contributed equally to this work, which was accepted to the 2024 Annual Network and Distributed System Security Symposium (NDSS).*

## 3.1 Overview

**Scenario.** In this thesis, we consider the widely-used *confidential-computing* scenario on the cloud, where the user wants to utilize the Intel SGX on the cloud to protect his/her data and applications. In this scenario, there are two entities: the cloud and the user. The user runs his/her applications inside enclaves, and wants to protect his/her data against side-channel attacks using PRIDWEN.

**Threat model.** Our threat model is similar to the threat models in other SGX-related studies[25, 49, 51]. Our TCB consists of an SGX enclave provided by an Intel CPU and everything inside the enclave, including PRIDWEN, a target Wasm binary prepared by the user, and the PRIDWEN pass selection policy. We assume that the user uses remote attestation to verify the validity of the CPU and PRIDWEN, and establish a secure channel with PRIDWEN to securely transmit his/her binaries. We assume that adversaries have already compromised the underlying privileged system software (*e.g.*, OS) to attack PRIDWEN and the target binary. Any threats due to potential vulnerabilities of the CPU and the code running insides enclaves are out of scope.

**Goals.** PRIDWEN is designed to achieve the following goals: *1) Adaptivity* PRIDWEN selects mitigation techniques that conform to the capabilities of the target execution platform on demand. PRIDWEN needs to optimally combine multiple mitigation techniques without

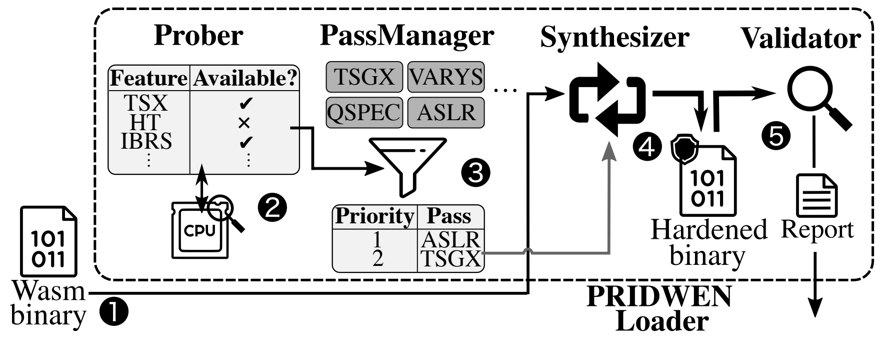

**Figure 3.1:** Overview of PRIDWEN. ❶ A user compiles a program into a Wasm binary and transmits it to PRIDWEN via a secure channel. ❷ PRIDWEN probes the hardware configurations. In this example, the CPU enables TSX and IBRS while disabling HT. ❸ PRIDWEN selects mitigation passes. Here, it chooses T-SGX and ASLR because the CPU enables TSX (for mitigating page-fault attacks) and IBRS (for mitigating Spectre variants). ❹ PRIDWEN synthesizes and hardens a native binary based on chosen passes. ❺ PRIDWEN validates the final synthesized native binary. A report is sent back to the user to attest the final binary.

causing conflicts or failures.

*2) Attestability* The second goal of PRIDWEN is to support the remote attestation of the dynamically generated binary inside SGX; Native SGX only supports attesting static binaries. PRIDWEN should allow users to verify the integrity of the final executable running inside an enclave, as well as obtain the genuine information regarding whether the executable is faithfully generated by PRIDWEN (*e.g.*, the selection and application of the mitigation schemes).

*3) Extensibility* Another goal of PRIDWEN is to be extensible, so that it can support forthcoming mitigation techniques against SCAs besides existing ones. Moreover, it should support multiple platforms due to the diversity of practical computing platforms. The extensibility of PRIDWEN should also allow for smooth integration of legacy mitigation techniques.

**Architecture.** Figure 3.1 shows an overview of PRIDWEN. The core of PRIDWEN is an in-enclave loader that implements key ideas with corresponding components: *user-mode hardware probing* (Prober), *optimal pass selection* (PassManager), *load-time program synthesis* (Synthesizer), and *post-synthesis validation & final binary attestation* (Validator). Given that each countermeasure may depend on specific hardware features, Prober interacts with the platform and dynamically determines the availability of these features. Based on the probing results, PassManager determines an optimal set of countermeasures (*i.e.*, instrumentation passes) and finalizes the order of applying them based on user policies. Next, PassManager informs Synthesizer about the final selection. Synthesizer takes a Wasm binary (provided by the user via a secure network channel) as an input and compiles it into a native one. During the compilation, Synthesizer hardens the binary with the optimal pass set provided by PassManager. Validator takes the synthesized binary as an input, and verifies that 1) each countermeasure is correctly enforced, and 2) no conflict exists among the enforced countermeasures. Validator also provides the functionality of attestation on the final binary.

## 3.2 PRIDWEN

### 3.2.1 Prober

The goal of Prober is to identify hardware capabilities of the target execution platform, which is needed by PassManager to determine the optimal set of mitigation schemes to enforce. This *hardware probing* step typically requires interactions with the system software, such as retrieving privileged registers (*i.e.*, Model-Specific Register (MSR) and control registers) and executing the `cpuid` instruction. However, we cannot rely on these approaches because the system software is not trusted in our threat model. SGX provides an attribute field called XSAVE-Feature Request Mask (XFRM) to determine whether some hardware features are enabled at enclave creation, but it only covers a few instructions (*e.g.*, AVX and MPX [116]). To solve this, PRIDWEN leverages exception handling and remote attestation to securely probe hardware configurations while running inside an enclave.

**Exception-based instruction probing.** The instruction probing identifies whether PRIDWEN can use hardware-assisted mitigation techniques relying on specific instructions. A CEH for SGX (§2.1) can be used to determine whether a target system supports or enables the required instruction. Specifically, the probing code executes the specific

```
#define UD 6 /* Invalid opcode exception */
bool tsx_support = false;
check_tsx_support:
  _xbegin();
    tsx_support = true;
  _xend();
exception_handler:
  if (SSA.GPRSGX.EXITINFO.VECTOR == UD &&
      SSA.GPRSGX.RIP == check_tsx_support) {
    GPRSGX.RIP = skip_tsx_check;
  }
```

**Figure 3.2:** Exception-based probing code for TSX. If a CPU does not support TSX, there will be a `#UD` exception that needs to be handled by an in-enclave exception handler to proceed execution (i.e., changing `GPRSGX.RIP`).

instruction demanded (*e.g.*, TSX) inside SGX, and then checks whether it results in a `#UD` exception by inspecting the exception information (Figure 3.2). If it does—i.e., the target platform does not support the instruction, PRIDWEN adopts a software replacement of the hardware-assisted mitigation if available, or omits it otherwise. The CEH then advances `GPRSGX.RIP` to continue execution.

Attackers can disrupt this type of probing, but it only results in Denial-of-Service (DoS) that they can always trigger without special attacks: 1) Attackers can simply resume the enclave execution right after the `#UD` exception without invoking the CEH. However, `GPRSGX.RIP` still points to the invalid instruction, and it is impossible to manipulate it or the exception number outside the enclave. Therefore, this only incurs repeated `#UD` exceptions. 2) Attackers can selectively enable a specific hardware instruction *only during probing* and disable it during the actual execution. This trick can deceive the probing, but it only introduces `#UD` exceptions during runtime, resulting in another DoS.

**Remote attestation for hardware configuration.** The remote attestation determines whether a target platform is vulnerable to SCAs that utilize certain hardware features. SGX remote attestation allows PRIDWEN to accurately determine several hardware configurations, i.e., HT and Indirect Branch Restricted Speculation (IBRS). If a remote device turns on HT, an attestation verification report will contain `CONFIGURATION_NEEDED` in the `isvEnclaveQuoteStatus` field since API version 3 [162, 163]. PRIDWEN can leverage this information to selectively adopt mitigations for preventing hyperthread co-locations [51,

**Table 3.1:** The APIs for the instrumentation. `CCTX`: `CompilerContext`. `MI`: `MachineInstr`. `MCTX`: `MachineContext`. `MB`: `MachineBasicBlock`.

| API | Hooking point |
|---|---|
| `onFunctionStart(CCTX *c)` | Beginning of a function |
| `onFunctionEnd(CCTX *c)` | End of a function |
| `onControlStart(CCTX *c)` | Beginning of a control statement |
| `onControlEnd(CCTX *c)` | End of a control statement |
| `onInstrStart(CCTX *c)` | Before a IR-level instruction |
| `onInstrEnd(CCTX *c)` | After a IR-level instrcution |
| `onMachineInstrStart(CCTX *c, MI *i)` | Before a native instruction |
| `onMachineInstrEnd(CCTX *c, MI *i)` | After a native instrcution |

50]. Also, if a remote device does not install the microcode update for indirect branch control mechanisms, a remote attestation protocol will indicate `GROUP_OUT_OF_DATE` [164]. If users still want to securely run their code on such an outdated device, they can adopt software-based approaches [108] against speculative SCAs. Without learning such hardware information, unnecessary performance overhead might be paid for applying redundant protections. It is worth mentioning that updating microcode and changing HT configuration require system reboot in general; As a result, malicious system software cannot manipulate these hardware configurations during the execution of hardened programs.

### 3.2.2 PassManager

PassManager is in charge of selecting and integrating multiple mitigation schemes. PassManager provides a set of high-level APIs that allows developers of side-channel mitigation schemes to implement their instrumentation passes and plug them into the PRIDWEN loader. During the load time, PassManager 1) maintains a list of plugged-in passes, 2) determines the optimal set of passes for Synthesizer to execute, and 3) resolves the correct application order of each selected pass to avoid conflicts.

**Pass APIs.** Table 3.1 lists the high-level APIs for implementing instrumentation passes. For instrumentation, we expose all the hooks as APIs. To reflect the structure of a Wasm module, we classify the IR-level hooks into the granularity of functions, controls, and instructions. Each hook can obtain the information about the hooking IR instruction and the current states of compilation via the `CompilerContext` (`CCTX`) data structure. For the native level, the hook should consult the information of the native instruction via the `MachineInstr` (`MI`) data structure, as the `CompilerContext` data structure does not track such information.

**Pass selection and ordering.** When being plugged into the PRIDWEN loader, each pass is associated with a configuration file that specifies the type of SCA to mitigate, hardware features or other passes that it depends on, and a list of passes incompatible with it. Each pass can also specify its weakly dependent passes, which indicates that the pass depends on these weakly-dependent passes only when they are available. During the initialization phase, PassManager adds all the plugged-in passes into a pass queue. For better flexibility, PRIDWEN also allows a user to customize the pass queue by providing a pass selection policy ($P$), which contains descriptions of all plugged-in passes and their dependencies. We detail our implementation of a pass selection policy in §3.3.2.

To select the optimal set of passes, PassManager takes the following steps: 1) It consults Prober about the current hardware configuration. 2) It checks the dependency of each pass in the queue, and drops a pass if the required hardware feature is not available. 3) It checks the types of side channels that each active pass mitigates; if PassManager identifies more than one passes targeting the same SCA, it retains the one with the highest priority value specified in $P$. If not specified, it will first assign a priority value to each of them based on several criteria including performance overhead. 4) To determine the application procedure of active passes, PassManager builds a dependency graph of all the passes given the dependencies specified in $P$. Next, PassManager uses the topological order of the graph as the application order. PassManager may drop passes if their strong dependencies are not satisfied or incompatible passes are in the active pass set[1]. If all passes are independent, PassManager uses the order in the pass queue as the application order. Here we assume the graph contains no circular dependencies; otherwise, PRIDWEN will terminate the execution.

[1]Note that we did not come across any SCA mitigations that are mutually exclusive. If such cases emerge in the future, programmers can specify this situation in the pass selection policy ($P$) and mark the involved mitigation passes as mutually exclusive; the pass with higher priority will be applied by default. Users may override the policy to choose a custom priority.

### 3.2.3 Synthesizer

Synthesizer uses *load-time synthesis* to dynamically generate a final binary hardened with the optimal set of mitigation passes (§3.2.2) for the current hardware configuration, and loads the binary into memory for execution. Synthesizer adopts a Wasm binary as the input, and takes three steps, *i.e.*, parsing, compilation, and instrumentation, to achieve this goal. We extend the compilation chain to support both IR- and native-level instrumentation so that it is flexible enough to integrate various types of SCA mitigation schemes with PRIDWEN.

**Parsing.** In the parsing step, Synthesizer performs standard decoding on a Wasm binary and converts it into Wasm IR. During decoding, Synthesizer also validates the format of the binary with several checks (e.g., type checking of functions) to guarantee that the binary follows the specification. Any modification to the binary before parsing can thus easily result in an immediate rejection. For example, inserting an instruction that causes the inconsistency on the stack machine renders the binary invalid.

**Compilation.** To generate the native binary inside the enclave given Wasm IR, Synthesizer performs a single-pass compilation over each function (similar to the baseline compilation of SpiderMonkey [99] and V8 [165]). During the compilation of a function, Synthesizer virtually executes each instruction based on the execution model of the Wasm stack machine and generates the corresponding native code. Synthesizer also keeps track of the metadata about each value (e.g., actual location and data type) on the operand stack to help correctly generate the native code and facilitate type-checking. In addition to the operand stack, Synthesizer maintains a control stack that keeps track of the control flow of the function. Pushing a value to the control stack indicates the function initiates a new control statement (e.g., `block`, `if`, or `loop` instruction), while popping a value from the control stack implies reaching the end of the current statement (e.g., an `end` instruction). The control stack provides sufficient information for Synthesizer to resolve the target of a branch (e.g., a `br` instruction). After finishing the native code generation of all functions, Synthesizer performs relocation. This process patches all the unresolved address values in the native instructions, such as `call` and those for memory accesses.

**Instrumentation.** To support flexible instrumentation, we extend the design of the compilation to provide hooks at both IR- and native-level. For IR-level hooks, we place them both before and after the position that Synthesizer processes an IR instruction to support code insertion, modification, and deletion. For each hook, we provide sufficient information about the corresponding instruction and the states of the compilation at the given point, such as the operands and the control stacks. Since Synthesizer may generate more than one native instruction for a single IR instruction, we provide similar hooks at the native level (i.e., surrounding the generation of native instructions) to support mitigation schemes that require the information about native instructions. To support the insertion or the modification of native instructions that require relocation, we provide the option to mark such instructions with symbols. A symbol refers to a target location that allows Synthesizer to recognize and resolve it during the relocation phase.

**Reproducible synthesis.** Since both the compilation and instrumentation are *deterministic*, Synthesizer has a nice property: the synthesis process is reproducible. This property ensures that given the same Wasm code and hardware configuration, the same version of PRIDWEN loader always generates the same final binary.

### 3.2.4 Validator

The flexibility of instrumentation indicates that an instrumentation pass can arbitrarily modify the binary. Such modifications can potentially disturb the already applied instrumentation passes or break the binary itself. To avoid such cases, Validator supports *post-synthesis validation* to validate whether the synthesized executable is hardened as expected. Also, since the runtime behavior of PRIDWEN cannot be determined beforehand, Validator provides *final binary attestation* to allow users to remotely verify both the process of synthesis and the hardened binary before execution. Both *post-synthesis validation* and *final binary attestation* are necessary to ensure the correctness of the final binary. In addition, they are conducted *only* once before running the program, and thus will not affect the runtime performance.

**Post-synthesis validation.** In post-synthesis validation, Validator conducts static analysis over a synthesized binary. Unlike typical binary analysis that assumes a stripped binary, post-synthesis validation enables more sophisticated analyses by taking advantage of the metadata (e.g., the control-flow information) provided by Synthesizer. Post-synthesis validation takes in the form of validation passes coupled with each instrumentation pass. Based on the control-flow information, Validator executes validation passes at the basic-block level. Validator iterates through all functions in the binary and invokes a procedure implemented in each validation pass at the beginning of each basic block, which performs a series of checks based on the content of the basic block (i.e., raw bytes). For example, a procedure can determine whether specific instrumentation is applied based on pattern matching. If any of the validation passes fails, Validator rejects the binary. Optionally, the procedure can utilize other metadata such as the original IR instructions that map to the basic block to facilitate the analysis beyond binary scanning.

**Final binary attestation.** In addition to using remote attestation for hardware probing (§3.2.1), PRIDWEN uses remote attestation to attest the dynamically synthesized binary inside the enclave. SGX does not natively support the attestation of dynamic enclave content; instead, traditional remote attestation measures only the static code and data that are initially inside an enclave, which is used as a piece of evidence throughout the process of remote attestation. To attest dynamic content, PRIDWEN incorporates a two-step scheme that extends the attestation of static content.

In the first step, a user uses the SGX standard procedure to attest the static part of PRIDWEN and establishes a secure channel [114]. Then, the user sends a Wasm binary `p.wasm` to PRIDWEN via the secure channel and PRIDWEN starts to synthesize the final binary based on the hardware configuration (`hw_config`) of the execution platform. In the second step, PRIDWEN sends the user: 1) the measurement of the synthesized binary `p.code`

**Table 3.2:** Attack surfaces and software-only or hardware-assisted mitigation schemes PRIDWEN implements. CPUs with recent microcode update do not have some of the attack surfaces.

| Attack surface | SW-only Mitigation | HW-assisted Mitigation |
|---|---|---|
| **Cache timing** | Interrupt (**Varys**) | Cache flushing (microcode) |
| **Page fault** | Interrupt (**Varys**) | **T-SGX** |
| **HT** | Co-location (**Varys**) | HT disabling (microcode) |
| **Speculative execution** | **QSpectre** | IBRS (microcode) |
| **Static layout** | **ASLR** | N/A |

(*i.e.*, the hash of native code blocks `hash(p.code)`) and 2) the `hw_config`. The user can then validate `hash(p.code)` thanks to a PRIDWEN's property: *reproducible synthesis* (§3.2.3). With the reproducible synthesis, the user can validate the final binary based on both `p.wasm` and `hw_config`.

## 3.3 Implementation

We implement a prototype of PRIDWEN with 25k lines of `C` code on top of the Intel Linux SGX SDK 2.5.102. The size of PRIDWEN in binary is only 1.26 MiB, maintaining a slim footprint of trusted computing base (TCB). For native code generation, we implement an `x86` backend to support the full Wasm instruction set.

**Runtime support.** Our prototype provides an Emscripten-compatible runtime support that allows to run fairly large, complex applications such as Lighttpd, as shown in §3.4. The application is directly compiled from unmodified `C` source code to a Wasm binary using the Emscripten compiler.

**Attestation of synthesized binaries.** Our prototype supports *final binary attestation* mentioned in §3.2.4. Also, the prototype provides a tool that allows users to locally validate the measurement of the synthesized binary.

### 3.3.1 Example Passes

To illustrate the capability and practicality of PRIDWEN, our prototype implementation integrates four SCA mitigation schemes (ASLR [109], Varys [51], T-SGX [49], and QSpectre [108]) into PRIDWEN using provided APIs, and *simultaneously* cover five important

SCA surfaces (cache timing, page fault, HT, speculative execution, and static layout) (Table 3.2) based on the probed hardware configuration (§3.2.1). Although the four mitigation schemes were not originally introduced by this work, they are selected to demonstrate how to integrate existing mitigation techniques using PRIDWEN. Because the control-flow information (including the definition of basic blocks in the binary) is required by most of the passes, we implement a control-flow analysis pass to share the information with other passes, eliminating the overhead posed by repetitive analyses. PRIDWEN helps to prioritize hardware-assisted mitigations with lower overheads if the execution platform supports them (e.g., TSX for T-SGX), and safely avoid redundant countermeasures if the platform is free from the corresponding SCAs thanks to recent microcode or hardware updates [46, 47, 166, 167].

*Example Pass #1: Fine-grained ASLR*

Many SCAs rely on accurate memory layout information to improve the granularity of leaked information. Thus, PRIDWEN enables fine-grained ASLR by default, which randomizes the location of every basic block, as a general mitigation scheme against SCAs.

**Integration.** We adopt the similar compiler-level scheme from SGX-Shield [109] by inserting a `jmp` instruction at the end of every basic block. First, the pass uses the `onControlStart` and `onControlEnd` APIs (Table 3.1) to identify the structure of a basic block. Next, the pass inserts a `jmp` with a symbol that points to the succeeding basic block if it does not end with a `jmp`. The pass also updates the targets of other branches accordingly by using the `onMachineInstrEnd` API. Later, Synthesizer shuffles the placement of each basic block if the ASLR pass is enabled. During the relocation, the generated symbols allow Synthesizer to resolve the target of each branch to a basic block at a randomized location.

**Post-synthesis validation.** We integrate a validation pass that performs the following checks: 1) whether a basic block terminates with a `jmp` and 2) whether each branch points to the correct target based on the control-flow information.

*Example Pass #2: T-SGX*

SGX allows the OS to handle page faults. Page-fault SCAs [25] exploit this design decision by intentionally making enclave pages inaccessible and observing which pages are accessed. To defeat this SCA, T-SGX [49] hides page faults from the OS by running an enclave spitted as small code blocks inside TSX transactions. As a result, all page faults occurring during the execution are suppressed (i.e., not delivered to the OS).

**Integration.** Our T-SGX pass has cache usage and execution time analyzers for native instructions using the `onMachineInstrEnd` API. Based on the analysis results, the pass determines the scale of a code block. Next, the pass replaces branch instructions at the end of the block with the instructions redirecting to the springboard (i.e., a `lea` for saving the address of the next code block and a `jmp` to the springboard). Similar to the fine-grained ASLR pass, the T-SGX pass identifies basic blocks in the binary by using the `onControlStart` and `onControlEnd` APIs. To support the springboard, the pass places the springboard code before the entry function (e.g., `main`) of the binary by using the `onFunctionStart` API.

**Post-synthesis validation.** The validation pass for T-SGX checks 1) the presence of the springboard, 2) the presence of the instructions to jump to the springboard at the end of every code block, and 3) whether the target of the instructions in step 2 correctly points to the springboard. Optionally, the pass can re-analyze cache usage and execution time to ensure the correctness of the code splitting.

*Example Pass #3: Varys*

SCAs usually require frequent interrupts or HT to accurately identify the execution context (e.g., which pages are accessed) or to attack in-core cache and speculation units. Varys [51] is a software-based approach to detect such behaviors.

**High-frequency AEX detection and cache eviction.** Varys identifies the occurrence and frequency of interrupts during the enclave execution by analyzing AEXs. Specifically, whenever an AEX occurs, SGX updates the corresponding field in the SSA. Thus, by counting the number of instructions executed at every basic block and periodically polling the SSA, Varys can estimate the frequency of AEXs. In addition, Varys explicitly evicts cache lines upon detecting AEXs to mitigate cache-based SCAs.

**Co-location test.** To prevent scheduling victim and attack threads to the same HT core, Varys prepares a pair of SGX threads and checks whether they are in the same core via an L1-cache-based covert timing channel. Varys performs this co-location test whenever it observes an AEX.

**Integration.** PRIDWEN's Varys pass inserts the checking code at the beginning of every basic block by using the `onControlStart` and `onControlEnd` APIs. Unlike the original Varys that counts the number the instructions at the LLVM IR level, our pass counts the number of native instructions with the help of the `onMachineInstrEnd` API. For the SSA polling routine, the pass inserts the code before the entry function of the binary via the `onFunctionStart` API. The pass also adds the co-location test code to the SSA polling routine (i.e., after detecting an AEX).

**Post-synthesis validation.** The validation pass verifies 1) the presence of the checking code, 2) the correctness of the instruction number added to the counter, 3) the presence of the SSA polling code, and 4) whether the target of the `call` in the checking code points to the SSA polling routine.

### *Example Pass #4: QSpectre*

One software-based approach to mitigate the Spectre attack is to use serializing instructions (e.g, `lfence`) to prevent the CPU from speculatively executing instructions beyond the intended placements. Following this idea, Microsoft Visual Studio has adopted a compiler-based scheme, QSpectre [108], which finds potentially vulnerable code patterns and inserts `lfence` instructions During compilation.

**Integration.** Instead of inserting `lfence` instructions based on pattern matching, which

```
[ { "name": "tsgx",
    "sca": [ "page" ],
    "dependency": { "hw": [ "tsx" ], "weak": [ "aslr" ] },
    "priority": "high"
  },
  { "name": "cotest-tsgx",
    "sca": [ "ht" ],
    "dependency": { "hw": [ "ht" ], "strong": [ "tsgx" ] },
    "priority": "high"
  },
  { "name": "qspectre",
    "sca": [ "spectre" ],
    "dependency": { "hw": [ "!ibrs", "ht" ] },
    "priority": "high"
  },
  { "name": "varys",
    "sca": [ "cache", "page" ],
    "dependency": { "hw": [ "!cache" ] },
    "priority": "medium"
  },
  { "name": "cotest-varys",
    "sca": [ "ht" ],
    "dependency": { "hw": [ "ht" ], "strong": [ "varys" ] },
    "priority": "medium"
  },
  { "name": "aslr",
    "priority": "high" } ]
```

**Figure 3.3:** Example pass selection policy $P$ (a `.json` file). `name`: name of the current pass; `sca`: the SCAs to mitigate; `dependency`: the required hardware (`hw`) and dependent passes (can be `weak` or `strong`); `priority`: the priority of the current pass.

can be bypassed [168], PRIDWEN's instrumentation pass for QSpectre adopts a simple, yet effective strategy: inserting `lfence` instructions to all `if-else` structures. More concretely, the pass inserts an `lfence` instruction right after the conditional branch in the code of an `if-else` structure. This pass uses the `onMachineInstrEnd` API and determines if a conditional branch is in an `if-else` structure by consulting the `CompilerContext` data structure.

**Post-synthesis validation.** The validation pass simply checks the presence of the `lfence` instruction in every `if-else` structure.

### 3.3.2 Pass Coordination

PassManager selects the optimal set of passes and resolves potential conflicts following the steps mentioned in §3.2.2. To incorporate the four passes into PRIDWEN, we specify the pass selection policy $P$ (shown in Figure 3.3). Note that the policy $P$ is just an example; PRIDWEN can transparently support any developer-provided policies by design.

**Pass selection.** First, PassManager selects all feasible passes according to hardware dependencies. For example, it selects the QSpectre pass if IBRS is disabled and HT is enabled (Line 13). Next, it prunes passes that target overlapping SCAs based on the `priority`. We prioritize T-SGX (hardware-based) over Varys (software-based) in $P$ such that if PassManager has selected the T-SGX pass because TSX is available, it will omit the Varys pass. Then, PassManager checks the unprocessed passes in $P$ and includes those whose dependencies are all satisfied (e.g., co-location test for T-SGX `cotest-tsgx`), or without any dependency (e.g., ASLR).

**Pass ordering.** Based on the dependencies specified in $P$, PassManager also determines the order of applying passes to resolve potential conflicts among them. For example, the ASLR and T-SGX passes can compete with each other to instrument branches at the end of basic blocks. To avoid such conflicts, a `weak` dependency between the two passes is indicated in $P$ (Line 3). Accordingly, PRIDWEN serves the ASLR pass first and then applies the T-SGX pass with slight modification (e.g., instrument `jmps` inserted by the ASLR pass to make it point to the T-SGX springboard). Also, PRIDWEN must apply the T-SGX co-location test pass `cotest-tsgx` after the T-SGX pass itself to correctly insert the testing code at the springboard, which is indicated in $P$ as a `strong` dependency (Line 8). As passes without dependencies (e.g., QSpectre) can be applied at anytime, PRIDWEN simply applies them after serving passes with dependencies.

## 3.4 Evaluation

We evaluate PRIDWEN on successful mitigation of individual targeted SCAs (§3.4.1), the semantic correctness of the input Wasm program (§3.4.2), the performance characteristics of the PRIDWEN loader (§3.4.3), and the performance overhead of PRIDWEN-synthesized binaries (§3.4.4).

**Experiment setup.** We ran all the experiments on a machine with a 4-core Intel i7-6700K CPU (Skylake microarchitecture) operating at 4 GHz with 32 KiB L1 and 256 KiB L2 private caches, an 8 MiB L3 shared cache, and 64 GiB of RAM. The machine was running Linux kernel 4.15. The PRIDWEN loader is compiled with gcc 5.4.0 and executed on top of the Intel Linux SGX SDK 2.5.102.

**Applications and test suites.** We use three real-world applications or libraries (Lighttpd 1.4.48 [169], libjpeg 9a [170], and SQLite 3.21.0 [171]) as a macro-benchmark suite representing large, complex applications, as well as a micro-benchmark suite, PolyBenchC [172]. The benchmark suite consists of 23 small `C` programs with only numerical computations (i.e., no `syscall`) that are used to evaluate the runtime performance of just-in-time compiled Wasm binaries against native C binaries [99]. We compile the original source code of each micro- or macro-benchmark program into Wasm using Emscripten [107], an LLVM-based compiler. We also directly port all of the programs using SGX SDK to serve as baseline versions. We use the official Wasm specification test suite [110] to test the correctness of the synthesis of PRIDWEN (§3.4.2).

**Methodology.** For each run of experiments, we take the compiled Wasm binaries as input to PRIDWEN. To evaluate PRIDWEN-synthesized binaries with distinct sets of defense schemes enforced, we manually configure PRIDWEN before each run. We use **BASE** to represent the configuration of baseline compilation (i.e., synthesis without instrumentation) and the name of defense schemes to represent the configuration of enforcing the corresponding schemes. For example, **TSGX** indicates the configuration with T-SGX enforced. For the ease of comparing Varys and T-SGX, the rest of the section uses **VARYS**, to represent its original design with the co-location test, respectively. **QSpectre** or **QS** represent QSpectre. To measure the execution time of each application, we use the `rdtsc` instruction via an `OCall` inside an enclave. The reported results are averaged over 10 runs.

### 3.4.1 Security Analysis

In addition to statically checking the enforcement of each mitigation scheme via validation passes, we also manually verify whether the integrated versions of example passes are effective and compatible by running simplified SCAs against them and building a test suite (§3.4.2). After hardening a test binary over the SCA surfaces with different combinations, we introduce frequent page faults and interrupts, manipulate processor affinity, and run a simple Spectre attack [20]. We confirm that the T-SGX pass suppresses page faults during runtime, the Varys pass detects frequent interrupts and thread co-location (if HT is enabled), and the QSpectre pass disrupts speculation (if IBRS is disabled). In addition, combining the ASLR pass and frequent interrupt detection (the Varys pass) can effectively mitigate or slow down an attacker's attempts to infer the fine-grained memory layout.

### 3.4.2 Correctness

To validate whether the synthesized program behaves as expected, we use the official Wasm specification test suite [110], which provides comprehensive test cases for all Wasm instructions. The test suite consists of 73 programs. Each program includes a set of functions and test cases that specify the expected outputs with given inputs. We ran the test suite on PRIDWEN with all hardening configurations and reported the results in terms of *pass* or *fail* on each program. In addition to the test suite, we also record intermediate values of all benchmark programs (by manually inserting `printf`) for both baseline and PRIDWEN-synthesized version and compare them.

**Results.** The results show that programs with all hardening configurations successfully pass all the test cases, which indicates that 1) the baseline compilation of PRIDWEN (**BASE**) faithfully follows the specification of Wasm, and 2) the enforcement of schemes does not modify the behavior of the program. Moreover, there is no difference when comparing intermediate values between **BASE** and the synthesized binaries.

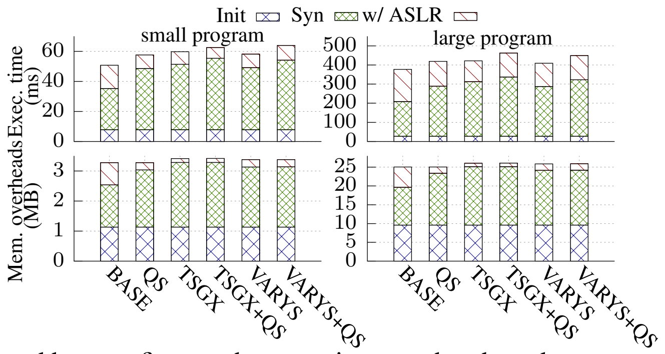

**Figure 3.4:** The top and bottom figures show runtime overheads and memory overheads, respectively, of PRIDWEN on program synthesis.

### 3.4.3 Performance of PRIDWEN

To show both runtime and memory overheads of the PRIDWEN loader, we measured the execution time that PRIDWEN takes to generate native C binaries and the additional memory that it allocates during the entire process (by hooking `malloc`). To demonstrate the impact on the size of input, we used one small (`2mm`, 52 kB) and one large (`lighttpd`, 462 kB) Wasm binaries as inputs. We also ran experiments with different configurations of PRIDWEN to show the impact of enforcing different mitigations. As the co-location test depends on either T-SGX or Varys and requires only adding a piece of code to each scheme, we do not include it the test in the selected configurations. Note that the overhead only needs to be paid once during the first initialization and synthesis.

The results are shown in Figure 3.4. We divide each bar into three parts: the initialization stage (blue), the synthesis stage (green), and additional overhead when the ASLR is enforced on top of the corresponding configuration (red). The initialization stage includes the time spent on hardware probing, PassManager initialization, and Wasm parsing. The synthesis stage represents the time spent on compilation and instrumentation in Synthesizer.

**Runtime performance.** For the runtime overhead (Figure 3.4), it is clear that given the same program, the execution time of the initialization stage is fixed regardless of the configurations. For the large program, PRIDWEN spends more time during the initialization stage, which is mostly due to the process of parsing the Wasm binary; however, the proportion of the execution time spent in the initialization stage decreases, which indicates that PRIDWEN spends more time on the synthesis stage for the large program. Also, enabling ASLR for the large program incurs higher overhead since it has more basic blocks. In addition, enforcing more schemes incurs higher overheads as expected. Overall, the one-time overhead of PRIDWEN is acceptable (less than 500 ms for the large program).

**Memory overhead.** For the memory overhead (Figure 3.4), the results show that PRIDWEN requires a fixed amount of memory during the initialization stage for the same program. PRIDWEN requires more memory for the large program, since the majority of the required memory is used to store the IR of the input program during the parsing process. We also observe similar memory requirements for PRIDWEN with the **BASE** configuration in the synthesis stage, since it needs more memory to maintain the metadata during compilation for the large program. Also, enabling ASLR on top of **BASE** incurs the highest overhead. The reason is that the instrumentation passes of each scheme all depend on the pass that manages the control-flow graph (CFG) information. As the ASLR pass can share the CFG information with other passes and can directly reuse such information during runtime, enabling ASLR on top of them incurs less overhead; when enabling just ASLR in **BASE**, it needs to generate the CFG information itself, resulting in a large memory overhead. Note that the memory overhead is only imposed once before the execution of the synthesized binary; thus it does not affect the memory usage of the binary in run-time.

### 3.4.4 Performance of Synthesized Binaries

We measure the runtime and memory overheads of PRIDWEN-synthesized binaries. We compare the results with those of native C binaries ported directly into SGX enclaves. In addition, we measure the performance overhead of the PRIDWEN-synthesized version of the defense schemes by comparing the results with those of the **BASE** configuration, and match it to that of the original implementations (i.e., the overhead indicated in the original papers). We confirm that the overheads are mainly inherited from the original design of the

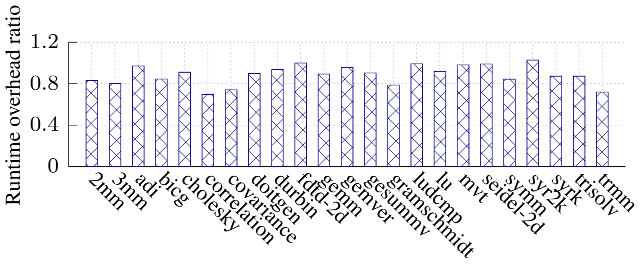

**Figure 3.5:** The runtime performance of PRIDWEN-synthesized Wasm programs compared to native C binaries.

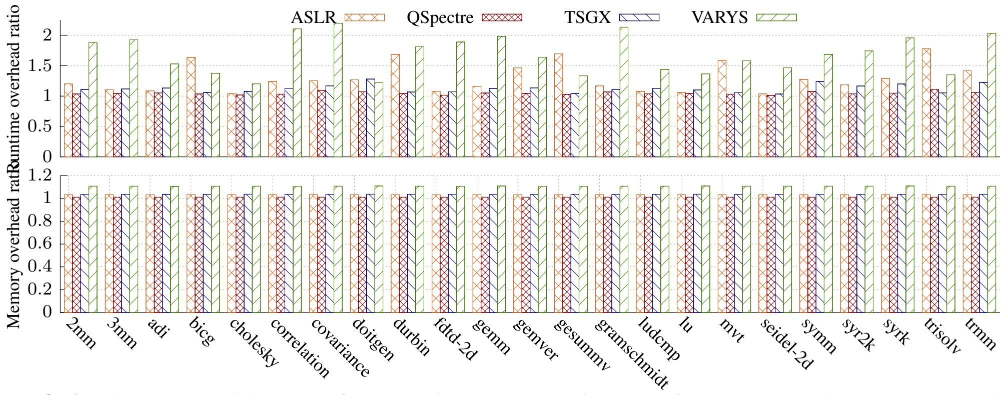

**Figure 3.6:** The top and bottom figures show the runtime performance and memory overheads, respectively, of PRIDWEN-synthesized programs secured with different mitigation schemes, compared to **BASE**.

countermeasures; PRIDWEN only imposes minimal amount of overheads in the binaries.

**Runtime performance.** Figure 3.5 shows the results of running the PolybenchC with the **BASE** configuration, which are normalized to the execution time of the native C programs. Our results indicate that PRIDWEN-synthesized binaries have negligible slowdown or are even faster than the native C binaries, without any mitigation schemes enforced. The execution time of PRIDWEN-synthesized binaries are 0.7×–1.0× of that of the native C binaries. Likewise, the very initial evaluation [99] of the in-browser Wasm compiler reports similar execution overhead results on PolybenchC programs (0.5×–1.4× of that of native C). The runtime performance of PRIDWEN-synthesized Wasm programs is comparable to or even better than that of C programs. This might be due to the small size of PolybenchC programs, with only numerical computations. Therefore, the synthesized Wasm programs are fairly compact and similar to native binaries. Furthermore, the difference between compilers (*i.e.*, Emscripten and GCC) may also contribute to the results. When programs get more complex, performance overhead increases as their Wasm forms no longer maintain the similarity to native binaries (*e.g.*, the number of instructions with one-to-many mappings grows).

Figure 3.6 demonstrates the results of running PolybenchC with defense schemes enforced. The bar on the figure represents the relative execution time of the program to the **BASE** configuration. **ASLR** incurs various overheads due to different numbers of randomized basic blocks being executed, which is not cache-friendly. Similarly, **QSpectre** also incurs various but smaller overheads, which result from the number of `lfence` instructions being executed.

Regarding T-SGX and VARYS, **TSGX** incurs less overhead than **VARYS** does. Especially, **VARYS** suffers from high overhead when it needs to check AEXs inside a loop structure. **VARYS** cannot avoid this issue without compromising its security guarantees. In contrast, **TSGX** supports loop optimization, which puts an entire loop into single transaction when possible.

**Memory overhead.** Figure 3.6 shows how much memory each mitigation demands on top of the binaries with **BASE**. On average, the memory overheads of the synthesized binaries are about 1.2× compared to the baseline binaries, which is moderate.

**Real-world applications.** We use three real-world applications as case studies to show that PRIDWEN provides sufficient support for large, complex programs. In addition to the *Lighttpd* (a web server), the other two applications are based on *libjpeg* and *SQLite* libraries. The *libjpeg* application supports both compressing and decompressing a *jpeg* image and the *SQLite* application supports basic database operations, including `insert`, `select`, `update`, and `delete`. We use the HTTP benchmarking tool, *wrk*, for evaluating the throughputs of the *Lighttpd*. For the other two applications, we measure the execution time of each supported operation and report the average values over 10 runs.

Figure 3.7 shows the results of *Lighttpd*. The slowdown of **BASE** is 1.5× to the native version. **TSGX** incurs less overheads compared to **VARYS** (1.9× versus 2.6×),

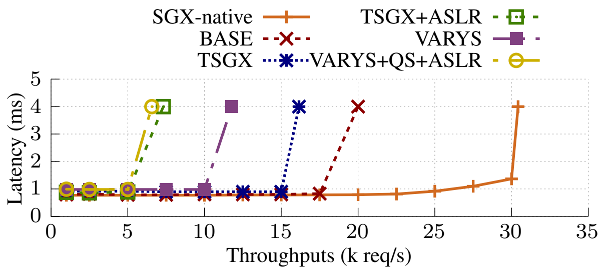

**Figure 3.7:** The performance of *Lighttpd*; QS: QSpectre.

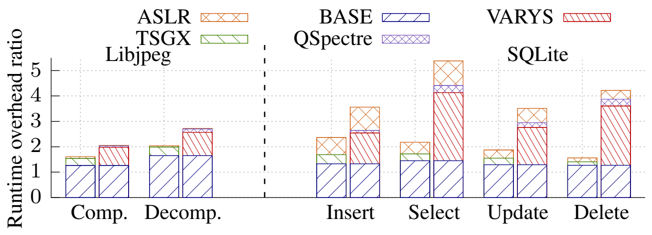

**Figure 3.8:** The performance of synthesized libjpeg and SQLite; each left bar illustrates hardware-assisted passes, while each right bar illustrates software-only passes.

demonstrating the advantage of hardware-assisted mitigation schemes over software-only ones. **ASLR** incurs significant overhead because *Lighttpd* has a large number of small-sized basic blocks; This shortens the gap between **TSGX** and **VARYS** when they are enforced with **ASLR**. When enforcing multiple mitigation schemes, the slowdown of *Lighttpd* is up to 4.6× compared to the native binary.

Figure 3.8 presents the performance of *libjpeg* and *SQLite* applications. We use stacked bars to represent the incurred overheads when applying the optimal set of mitigations on top of a viable hardware configuration. The runtime overhead of **BASE** is 1.2×–1.7× compared to the native versions. The overheads of individual mitigation schemes are similar to the results of PolyBenchC presented in Figure 3.6 (e.g., **TSGX** incurs less overheads than **VARYS**). The average slowdown of hardware-assisted mitigation schemes is 1.9× while that of software-only mitigation schemes is 3.4×. Depending on hardware configurations, hardware-assisted mitigation schemes are 2.1×–5.4× faster than software-only ones.

Again, as PRIDWEN only aims to integrate multiple mitigation techniques, most of the performance overheads are inherited from the original design of the countermeasures, while PRIDWEN itself does not impose significant overheads.

## 3.5 Discussion

**Adding new defenses to PRIDWEN.** PRIDWEN is designed with future scenarios in mind. Thanks to the high-level instrumentation APIs provided by PRIDWEN (Table 3.1), new instrumentation-based defense techniques can be easily added to PRIDWEN as a pass. PRIDWEN pass APIs offer different instrumentation granularity with sufficient information about the corresponding instruction on both IR- and native-level, which should meet all instrumentation needs. We recommend to develop new defenses directly with PRIDWEN to save the hassle of replacing heterogeneous instrumentation APIs with PRIDWEN versions during integration. The developer will also need to provide the type of side-channel attack targeted by the new defense, and the possible dependency on specific hardware features or existing techniques in the pass selection policy. This allows Prober and PassManager to resolve potential incompatibility issues and conflicts, and correctly enforce the new defense. Compared to the effort needed to write a new pass, writing a pass selection policy should be much simpler.

**Upgrade PRIDWEN.** Modern hardware is evolving quickly with updated features useful for security purposes and PRIDWEN is designed to keep up with the hardware evolution. It is necessary for PRIDWEN to allow probing of new hardware features for defense techniques built with such features. The probing logic for the specific hardware feature will need to be added by PRIDWEN developers using either exception-based instruction probing or trusted remote attestation (§3.2.1) to bypass untrusted privileged software. Novel and secure techniques are also welcomed to probe the hardware capability of the platform. We hope that PRIDWEN can motivate hardware manufacturers to provide official secure probing helpers for hardware features. Upgrade of PRIDWEN is the effort of both the hardware and the software communities. The power of PRIDWEN is truly unleashed when equipped with the most updated inclusion of hardware features and mitigation schemes.

**The recent SmashEx attack [173].** A recent paper presents the SmashEx attack targeting the SGX SDKs; the targeted SDKs do not properly handle *re-entrancy* in their asynchronous exception handling logic, which allows the attacker to compromise the integrity of the SSA region and modify `GPRSGX.RIP`. PRIDWEN is built on top of the trusted Intel SGX SDK. The vulnerability of asynchronous exception handling in SGX (SmashEx) is rooted in the SGX SDK, and it was already mitigated by recent patches [174, 175]. That is, the enclave-specific SSA that hosts the GPRSGX.RIP cannot be compromised with the attack in the latest SDK. PRIDWEN should work with the latest SGX SDK because it does not heavily rely on or modify a certain SDK version.

**Program synthesis on the cloud side vs. the user side.** PRIDWEN makes the design decision of conducting program synthesis on the cloud side to minimize the extra effort required for preparation on the user side. An alternative solution would be deploying a dedicated program on the cloud to report the hardware configuration back to the user, then the user in return synthesizes and caches the hardened binary themselves and sends the binary to the cloud for deployment. Synthesizing and caching the binaries of different configurations at the user side can save compilation cost on the cloud side; however, this puts more burden on the user. In addition, it would be difficult to update the binaries on the cloud side when the configuration changes, since the server has to wait for the user to compile and transmit the updated version. In contrast, when the configuration on the server is changed (*e.g.*, after reboot), PRIDWEN automatically re-synthesizes the application and applies the change upon restarting. It is also possible to optimize PRIDWEN in a similar way by caching the synthesized programs of different configurations on the cloud side to reduce the compilation overhead.

# CHAPTER 4
# SENSE: ENHANCING MICROARCHITECTURAL AWARENESS FOR TEES VIA SUBSCRIPTION-BASED NOTIFICATION

## 4.1 Overview

While SENSE can be a generic mechanism for feeding microarchitectural events directly to userspace TEEs, in this thesis, we initially demonstrate how to thwart cache-based SCAs—the most prominent type of SCAs in TEEs—with SENSE.

### 4.1.1 Targeted Platforms

SENSE targets platforms with a TEE component deployed and a modern cache architecture implemented.

**Cache architecture.** We assume a standard modern cache architecture, featuring multiple cache levels, including core-exclusive (L1 and L2) and shared (LLC) caches. Core-exclusive caches undergo flushing during context switches (which is aligned with most recent TEE architectures [176, 177, 61]). Additionally, we assume the cache controller is configured and only configured via dedicated registers, which is also consistent with common platforms.

**Types of TEEs.** SENSE is designed to accomodate both process-based TEEs (*e.g.*, Intel SGX [15]) and guest-user processes within VM-based TEEs (*e.g.*, AMD SEV [17] and Intel TDX [16]). From an architectural perspective, memory accesses differ between processes and VMs in that the latter must additionally traverse the extended page tables (EPTs). Leakage from EPT walks are out of scope for this work, and consequently the SENSE ISA extension (§4.2.2) remains consistent across both processes and VMs. SENSE is not applicable to ARM TrustZone [18], which is a root TEE.

**Trusted TEE component.** TEEs provide established mechanisms to safeguard sensitive code within secure execution contexts known as enclaves. A trusted software component enforces access control mechanisms to maintain separation between secure enclaves and untrusted domains [15, 176, 61, 177]. Additionally, in the case of SENSE, the trusted software component handles other security-sensitive operations, such as determining the TEE status (*i.e.*, whether the thread is under TEE or not) and setting up cache event monitoring and notification mechanisms, as detailed in Section §4.2. We assume that security-critical metadata containing the TEE status is transmitted alongside memory requests. These assumptions align with current academic [176, 177, 178, 61] and industrial solutions [15, 179, 180].

**Characteristics of TEE programs.** We assume that the portion of isolated execution constitutes a minority of the application workload, as TEEs are most effective when the isolated execution is minimized to reduce the attack surface. This aligns with the intended usage of TEEs, where only small, sensitive code components are allocated to the TEE [57, 59, 60]. Although SENSE operates correctly even if the majority of the workload is isolated, the performance of isolated execution may be impacted [57, 15]. In addition, we assume that sensitive code uses writable shared memory solely for I/O, if at all, and access patterns to this shared memory do not leak information. Isolated code should focus on processing local data, limiting I/O needs to copying input and output data in and out of the component.

### 4.1.2 Threat Model

We adopt a robust adversary model aligned with hardware/firmware-defined TEE architectures [25, 49, 51], where both the OS kernel and hypervisor are considered untrusted. The adversary can execute access-based [7, 139, 140, 141] and conflict-based [4, 136, 6, 5, 137, 138] cache SCAs to extract information from a sensitive execution domain (*i.e.*, enclave). The adversary can initiate attacks from all privilege levels (excluding the highest level containing the trusted software component), access precise timing measurements and eviction instructions, and launch attacks from the same or a different CPU core.

**Out of scope.** We do not consider physical attacks on caches [181], fault injection attacks [182], or hardware flaw exploitation [183, 184, 185]; nor do we consider denial-of-service attacks from a security perspective. We assume that the adversary cannot compromise the trusted software component.

### 4.1.3 High-level Approach

The core of SENSE is an architecture-agnostic processor extension that features a new CPU SENSE *mode*, which notifies a running thread inside TEEs of the events it subscribes to. Only threads operating under TEEs run in SENSE mode and can subscribe to relevant microarchitectural events.

SENSE assumes that any TEE that is leveraged is in compliance with its original design intent, meaning that the majority of the execution workload is not security-critical and only a smaller portion is security-critical and isolated within TEEs (§4.1.1).

SENSE consists of three modules organized by functionalities. The *Subscription Module (SM)* (§4.2.1) enables the subscription of cache-related microarchitectural events (e.g., cache evictions) for threads running in TEEs and monitors the occurrence of these events. The *Notification Module (NM)* (§4.2.2) manages the CPU SENSE mode and provides the architectural interface of delivering microarchitectural events (*e.g.*, cache events) to userspace TEEs for handling. The *Action Module (AM)* (§4.2.3) allows the occurred events to be handled by a userspace event handler within TEEs.

**The benefit of SENSE.** SENSE is a novel interface for userspace applications in TEEs to listen to fine-grained microarchitectural events and take corresponding actions. Compared to detection-based defenses against SCAs (§2.4), SENSE has the following benefits:

*1) Trusted native information source.* By letting the hardware directly notifying the TEEs, SENSE bypasses the untrusted privileged software to provide microarchitectural information, fitting the threat model of TEEs. The control flow transition to the userspace handler is enforced by the CPU without changing privilege levels, ensuring that the handler operates within the same enclave of the program.

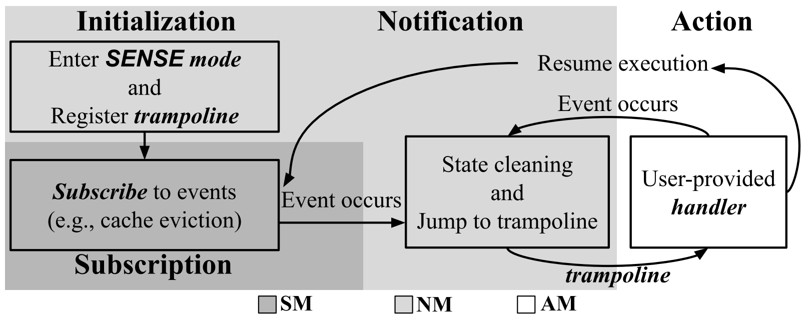

**Figure 4.1:** SENSE high-level workflow.

*2) Prompt and precise notifications.* SENSE notifies the userspace program about the events of the exact subscribed cache entries as soon as they occur, guaranteed by the existing communication channels of memory requests. By receiving instant notifications about cache events, victims can react promptly to potential cache SCAs, reducing the window of opportunity for attackers to leak secrets.

*3) Versatility and extensibility.* The subscribe-notify-act mechanism empowers victims with the ability to dynamically adapt their defenses based on real-time cache events, leveling the playing field with attackers who continuously evolve their strategies. SENSE also supports custom event handlers. If necessary, developers can define their own handlers that suit their own applications best. Moreover, by exposing microarchitectural events, many applications beyond SCA defenses are made possible (§4.2.5).

*4) Compatibility and generality.* SENSE is compatible with any existing partitioning and randomization-based cache SCA defenses without incurring significant extra hardware overhead. Besides, the NM is orthogonal to specific processor architectures, while the SM is cache organization agnostic. Moreover, SENSE is backward compatible and does not affect programs running without TEEs (§4.2.4).

*5) Security.* As SENSE also falls into the attacker's arsenal, SENSE should not open new attack surfaces (§4.3).

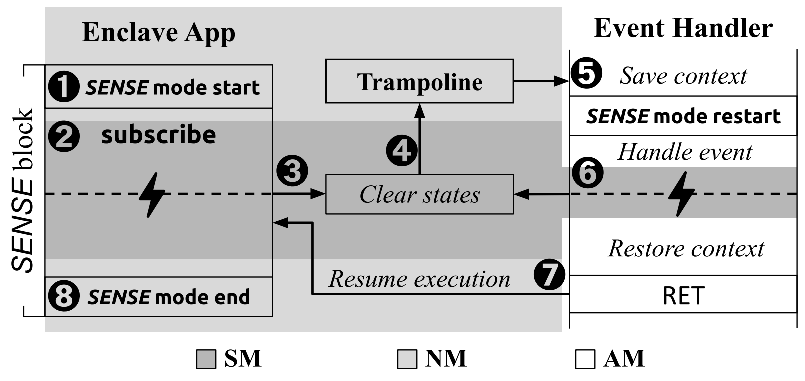

**Figure 4.2:** SENSE in action.

## 4.1.4 SENSE in Action

A closer look at the SENSE notification flow and the collaboration among the three modules at runtime is shown in Figure 4.1 and Figure 4.2.

*1) Initialization (NM):* The enclave thread enters SENSE mode at the beginning of the enclosed security-critical block (❶). The address of the *event handler trampoline* inside the enclave is registered with the CPU for event notification. The trampoline is an address within the enclave where a *userspace event handler* will be invoked.

*2) Subscription (SM):* Cache microarchitectural events (*e.g.*, cache evictions) for the memory resources within the enclave are subscribed under SENSE monitoring (❷).

*3) Notification (from SM to NM):* During the execution of an instruction at `RIP`, a subscribed event occurs (*e.g.*, a monitored cache entry is evicted). The SENSE mode pauses and the microarchitectural states related to subscribed events are cleared (*e.g.*, flush the cache) (❸). The control flow will be transferred by the CPU to the provisioned *event handler trampoline* inside the enclave (❹). If the event is an interrupt or exception, the control flow will be transferred to the event handler trampoline after the interrupt or exception is served by the OS. Detailed microarchitectural information about the event that occurred (*e.g.*, the thread identity that causes the event) is also supplied to the enclave. The trampoline pushes the `RIP` onto the user stack for later return and jumps to the SENSE handler within the enclave.

*4) Action (from NM to AM):* The SENSE handler saves the interrupted context, including the volatile registers and flag states to the enclave stack, and reenters SENSE mode. The handler then handles the event and restores the saved context before returning to `RIP` (❺). If another event occurs during the handling of the current event (❻), nested handling occurs by repeating steps ❸ – ❺.

Finally, the enclave thread resumes at `RIP` after the event is handled (❼). The enclave thread exits SENSE mode completely when it finishes, and then all subscriptions are canceled (❽).

## 4.2 SENSE Design and Implementation

In this section, we provide design details of the three modules in SENSE, as summarized in the architectural overview in Figure 4.3. Different implementations can be adopted for the SENSE design. To demonstrate the practicality of SENSE, we detail the prototype implementation of SENSE on a cycle-accurate gem5-based [111] `x86` emulator with its out-of-order (O3) CPU model and the default Classical Memory System.

### 4.2.1 Subscription Module (SM)

Extensions to the existing cache architecture are required to precisely locate a specific cache event and promptly deliver the notification. In this section, we discuss two SENSE cache extensions: *precise subscription*, for locating the exact subscribed events, and *prompt delivery*, for initializing immediate notifications. The extended SENSE cache entry is shown in Figure 4.3 and the cache controller logic for SM is depicted in Figure 4.4.

**Precise subscription of events.** *Precise subscription* allows users to subscribe to the occurrence of events at the *exact* cache entry monitored by SENSE, instead of monitoring the entire cache component.

*Requirements.* Cache entries to be monitored should be precisely annotated when they instantiate in the cache. Depending on the structure of the cache, the SENSE annotation could appear as an extra bit indicating whether the cache entry is monitored under SENSE

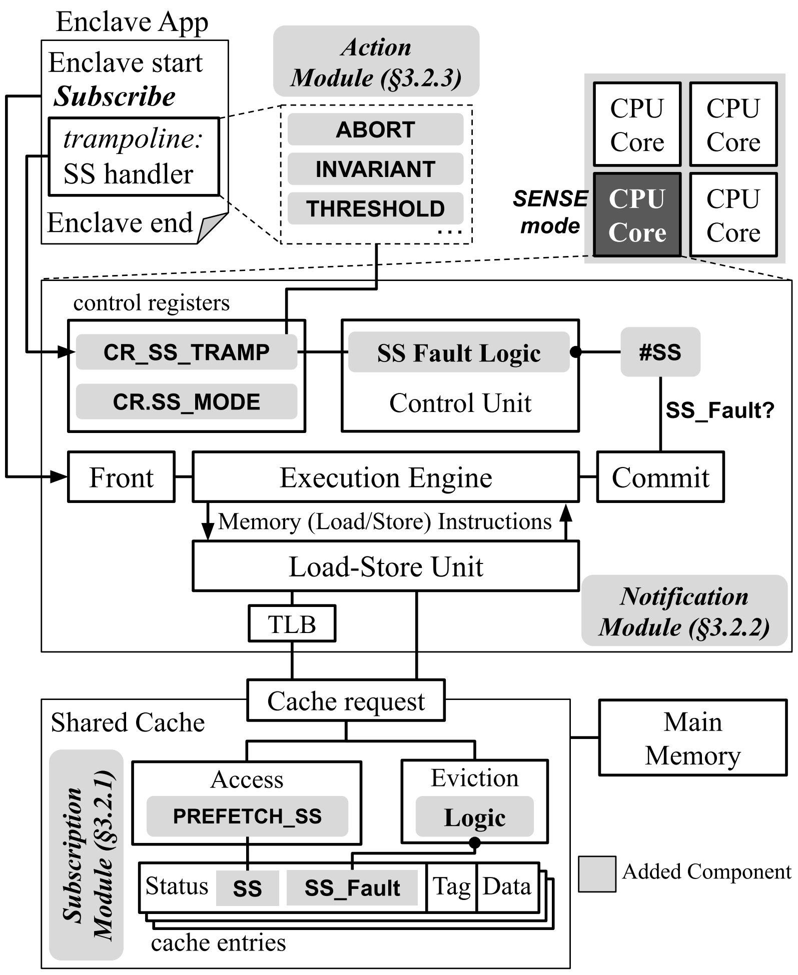

**Figure 4.3:** SENSE architecture: Subscription Module (SM) (§4.2.1), Notification Module (NM) (§4.2.2), and Action Module (AM) (§4.2.3).

or as a hash map that records the information, for instance.

*Design.* An entry in cache will be annotated for SENSE status, preferably using an unused reserved bit to avoid extra hardware overhead. When a cache event of interest involves an SENSE-annotated entry, the cache component should raise a notification flag, indicating the need for the CPU under SENSE mode to deliver the notification for handling.

**Prompt delivery of events.** *Prompt delivery* ensures that the CPU under SENSE mode will immediately start delivering the notification to the enclave when a subscribed event occurs.

*Requirements.* An information channel needs to be established to promptly inform the NM about the raising of a notification flag in the SM. Fitting such a channel into the existing

CPU microarchitecture is non-trivial. The channel should be close to the occurrence of the event, both temporally and spatially, to promptly relay the raised notification flag in a timely fashion. Meanwhile, the addition of the channel should not affect the correctness of existing microarchitectural components or impose too much performance overhead.

*Design.* When executing a memory instruction (*i.e.*, load or store), the CPU Load-Store Unit (LSU) first consults the TLB for address translation and then sends a load or store request to the cache. The status of the notification flag will be embedded in the existing responses sent back to the CPU LSU from the cache using reserved data space to avoid extra hardware overhead. When finalizing the memory responses, the CPU will start delivering the notification if the notification flag is set. Such a signal piggybacked onto the existing memory communication channel is aligned with the real-time workflow of the memory microarchitectural components and thus can immediately start delivering the notification as soon as a memory request triggers a monitored cache event.

**Implementation.** Achieving *precise subscription* and *prompt delivery of events* in the gem5-based `x86` emulator incorporates five major tasks. Note that the tasks are not specific to gem5 but applicable to general modern CPU architecture.

*1) Adding extra* SENSE *status bits in each cache entry.* SENSE chooses to extend cache entries with extra bits. Specifically, each cache entry is extended with two SENSE status bits: `SS` bit and `SS_FAULT` bit (Figure 4.3). The `SS` bit indicates whether the cache entry is monitored under SENSE mode, while the `SS_FAULT` bit represents whether the cache entry has been evicted when monitored under SENSE. Both SENSE status bits are `off` (*i.e.*, `0x0`) for all cache entries when booting up.

*2) Initializing cache event subscriptions.* In SENSE, subscribing to microarchitectural events occurs at the beginning of the enclave process. For cache eviction events, monitoring relevant cache entries under TEEs is realized by prefetching the entries into memory and turning on the `SS` bit. Therefore, similar to the `prefetch` macro-op, the subscription request boils down to a load (`ld`) CPU micro-op and is additionally extended with a `PREFETCH_SS`

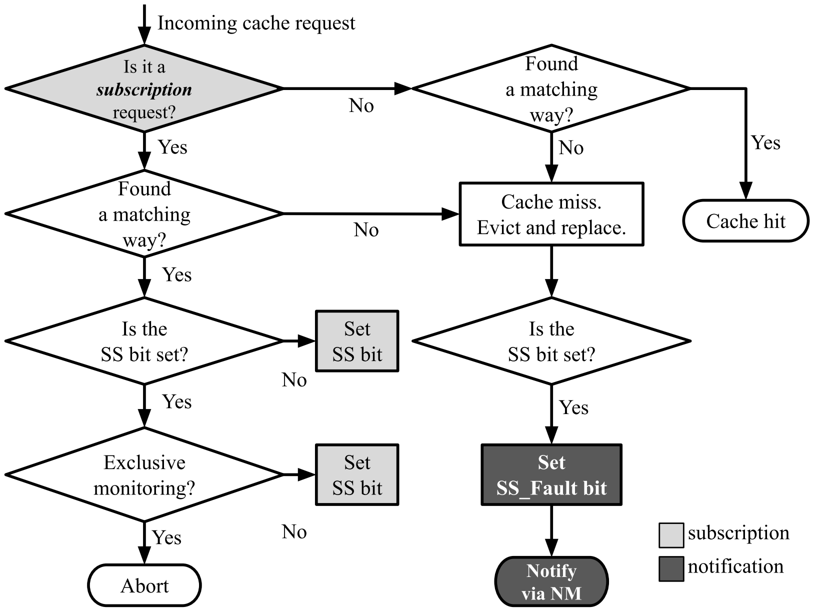

**Figure 4.4:** SENSE cache controller logic.

memory request flag to indicate the intention of event subscription under SENSE. At the beginning of the enclave process (*e.g.*, enclave initialization), the cachelines corresponding to the sensitive data are prefetched into all levels of the cache hierarchy.

*3) Marking the cache entries under* SENSE *monitoring.* Load (`ld`) instructions enter the Load-Store Queue in the CPU execution engine when pipelined, and are executed by the CPU LSU. After the CPU LSU forwards the memory loading request to the corresponding microarchitectural component, *i.e.*, cache, `PREFETCH_SS` flag is checked while performing the memory loading operation. If the `PREFETCH_SS` flag of the memory request is set (*i.e.*, due to the memory prefetching under SENSE), the `SS` bits of the prefetched cachelines are turned *on* (*i.e.*, `0x1`) (Figure 4.4), meaning they are *watched* under SENSE mode. The set of cachelines under watch for the current TEE thread is called the *watch set*.

*4) Signaling the occurrence of a subscribed eviction.* Traditionally, during the eviction of a cache entry, either due to active flush (*e.g.*, by using `clflush` instruction) or Least-Recently-Used (LRU) status caused by entry conflicts, the entry will be cleared and invalidated. If any member in an enclave thread's *watch set* (*i.e.*, `SS` bit is set for the cache entry) is evicted while the enclave thread is executing under SENSE mode, the `SS_FAULT` bit will be set for the entry

**Table 4.1:** SENSE new ISA instructions.

| **Instruction** | **Description** |
|---|---|
| `ssbegin [addr]` | Start SENSE mode with trampoline at `addr` |
| `ssend` | Terminate SENSE mode |
| `sstramp [addr]` | Push `rip` and transfer to `addr` after preparation |
| `sssub [addr][type]` | Subscribe to `[type]` event of resource at `addr` |
| `ssunsub [addr][type]` | Unsubscribe to `[type]` event of resource at `addr` |

(Figure 4.4), indicating an event of interest has occurred and the NM should be informed to immediately deliver a notification. When responding to the load/store request from the CPU LSU, the cache will check the `SS_FAULT` bit of the entry corresponding to the request. *5) Promptly informing the NM.* If the `SS_FAULT` bit is set, the cache will flush all entries to eliminate cache microarchitectural traces and add an additional `SS_FAULT` flag to the memory request response. As soon as the CPU LSU receives the response, if the response has the `SS_FAULT` flag set, the CPU will enter the NM and trigger an SENSE notification (detailed in §4.2.2) to the enclave thread using the information in the response. Eventually, the event handler trampoline starts execution, indicating a cache eviction event has occurred. When the enclave process finishes execution, the SENSE status bits of the entries in the *watch set* are turned `off`, and the *watch set* is reset.

### 4.2.2 Notification Module (NM)

**CPU SENSE mode and new ISA instructions.** A user-level enclave program under SENSE mode will get notified if a subscribed microarchitectural event has occurred. Upon notification, the CPU will supply detailed microarchitectural information to the enclave and jump to a registered *event handler trampoline*, where a *userspace event handler* will be invoked. By default, under SENSE mode, all interrupts and exceptions will be notified. An enclave process is also allowed to subscribed to cache microarchitectural events (*e.g.*, cache evictions) related to specific memory resources in the enclave. The SENSE architecture provides several new ISA instructions, shown in Table 4.1.

**SENSE block and event notification.** Any sequence of enclave instructions executed under SENSE mode is referred to as an SENSE *block*. If any subscribed event occurs during the execution of the SENSE *block*, the enclave thread will be notified. Specifically, an enclave thread is *notified* of subscribed SENSE events by noticing the control flow transfer to the *event handler trampoline*. This occurs when one of the following two conditions is met:

- A subscribed event is detected on the logical processor where the SENSE-protected enclave thread is executing;
- The enclave thread invokes an instruction that makes the SENSE architecture unable to track events for the thread (*e.g.*, `SYSCALL`).

Upon a notification, the SENSE mode will pause (*i.e.*, temporarily suspend delivery of SENSE notifications), and the microarchitectural states of the subscribed events will be cleared (*e.g.*, flush the cache), which aligns with the strategies adopted by existing TEEs [176, 177, 61]. Pausing SENSE mode when an event occurs is necessary to safeguard the interrupted context information from being overwritten. This precaution prevents potential issues arising from an immediate subsequent event landing at the beginning of the handler, which is responsible for preserving and restoring the execution context (§4.2.3). If an enclave thread running under SENSE mode is interrupted or triggers an exception, execution will resume at the *event handler trampoline* after the interrupt/exception has been serviced by the OS. This feature of SENSE is essential because microarchitectural events for the enclave thread cannot be handled by userspace handlers under untrusted kernel context. Therefore, when a thread resumes execution, the event handler should reset the microarchitectural state for all events that were being tracked at the time the notification was delivered.

**Modes of monitoring.** As SENSE does not track the identity of the subscribing enclave in the cache entries, when enclaves on different cores monitor the same event, the event notification will be *broadcast* by default, *i.e.*, delivered simultaneously to all subscribing enclave threads. This broadcasting strategy provides the convenience of sharing information about an event that might be of interest to multiple enclaves, while it may open up the attack surface due to improper synchronization and interaction among different event handlers (discussed in §4.3). To ease the effort of designing SENSE applications, SENSE offers the option to enforce *exclusive monitoring* of events in a *first-come, first-served* manner, which can be useful for applications that emphasize security. If an event is being exclusively monitored, only the enclave thread that subscribes to the event first will receive SENSE notifications; the subsequent subscription requests from other threads are aborted (*e.g.*, exception), thus thwarting any potential interference among different event handlers.

**Implementation.** We implement the NM in our prototype by integrating extended control registers and modifying the existing interrupt controller logic, enabling hardware interrupts without privilege level changes, providing a streamlined and efficient mechanism for event notifications.

SENSE *control registers (CRs).* We equip each CPU logical core with a `CR.SS_MODE` control register bit and a `CR_SS_TRAMP` control register (Figure 4.3). Implementing this upgrade incurs zero hardware overhead when utilizing unused bits (*e.g.*, `CR4` bit 26-63 in `x86`) and reserved control registers (*e.g.*, `CR5-7` in `x86`), if available. The `CR.SS_MODE` bit holds the status about whether the CPU core is currently under SENSE mode. When an event occurs, the CPU will check the `CR.SS_MODE` bit and decide whether to deliver a notification. The `CR_SS_TRAMP` register hosts the address of the userspace *event handler trampoline* (§4.2.2), the location where the control flow lands after the CPU decides to deliver the notification. We implement the new ISA instructions (Table 4.1) as follows:

- `ssbegin`: This instruction is invoked by the trusted software component of the TEE at the beginning of the enclave program. The `CR.SS_MODE` bit will be set to indicate that the CPU enters SENSE mode if and only if the running thread is under TEEs (*e.g.*, determined by the trusted software component), and the address of the reserved *event handler trampoline* will be loaded into the `CR_SS_TRAMP` register.
- `ssend`: This instruction is invoked by the trusted software component of the TEE at the termination of the enclave program. The `CR.SS_MODE` bit will be unset to stop the delivery of notifications if the finishing thread is under TEEs, and the `CR_SS_TRAMP` register will be cleared, terminating the SENSE mode completely.

- `sstramp`[1]: This instruction is invoked inside the trampoline. It pushes the current `rip` for later return and transfers control to the user handler.
- `sssub`: This instruction emits a subscription request detailed previously in §4.2.1. The cacheline corresponding to `addr` is prefetched into all levels of the cache hierarchy.
- `ssunsub`: This instruction clears the monitoring status set by `sssub`.

*Event notifications in the form of* SENSE *Faults.* The delivery of a notification takes in a form of a dedicated hardware exception triggered by the CPU using NM, minimizing the modification to the existing hardware architecture. We define such a hardware exception as SENSE *Fault (#SS)*, which occupies an unused interrupt/exception vector (*e.g.*, 20 under `x86`) of the CPU. Supplemental information can be included in the fault and passed to the handler for handling, such as the thread identity that causes the event.

SENSE *Fault control logic.* When an SENSE Fault occurs, instead of vectoring into a preregistered hardware exception handler in the interrupt vector table, the SENSE thread is trapped into a dedicated piece of SENSE Fault handling logic in the CPU control unit (Figure 4.3). The SENSE Fault control logic performs the following decisions and operations. Under SENSE mode, indicated by the `CR.SS_MODE` bit, the CPU will first pause SENSE mode by unsetting the `CR.SS_MODE` bit. Then, the CPU will check whether the the `CR_SS_TRAMP` register holds a valid address by testing the address. Finally, the CPU will extract the supplemental information corresponding to the event supplied in the SENSE Fault, push the data to the enclave stack, and transfer the control flow to the address specified in the `CR_SS_TRAMP` register. Note that the control flow transition to the userspace handler is enforced by the CPU without changing privilege levels, ensuring that the handler operates within the same enclave of the program.

**Software support.** We provide wrapper functions with function signature `_func()` for SENSE ISA instructions for ease of use. A code snippet for basic software usage is shown

[1]`entramp` is extendable for custom prepossessing operations before jumping to the handler, such as those inserted by the compiler but omitted by the hardware control flow transfer. For example, `entramp` can be extended to optionally decrementing `RSP` for `n` bytes to protect the stack red zone [186].

```
void handler(int ss_info) {
  /* Implement policies.*/
  if (ss_info == 1) {
    /* Handle accordingly */
  }
}

/* _ssbegin(handler) by TEE
      at enclave entry */
void enclave_program() {

  /* watch resource at ptr
  with size and type */
  _sswatch(ptr, size, type);
  ...
  /* Event occurs */
  /* Resuming point */

  /* unwatch resource at ptr */
  _ssunwatch(ptr);
}
/* _ssend() by TEE
      at enclave termination */
```

**Figure 4.5:** An example of SENSE in an enclave application.

```
/* Pseudo code
      of _ssbegin() */
__inline void _ssbegin() {
  __asm {
    trampoline:
    // landing point
    sstramp sshandler

    sshandler:
    call save_cpu_states
    ssbegin trampoline
    mov $rdi, $rax
    call g_handler
    restore cpu states
    ret

    entry:
    // Update the ptr
    g_handler = handler
    // save trampoline
    ssbegin trampoline
  }
}
```

**Figure 4.6:** Pseudo code of `_ssbegin()` function.

in Figure 4.5. The SENSE block is defined by wrapping the enclave code section with a pair of `_ssbegin()` and `_ssend()` functions by the trusted software component of the TEE at the enclave entry and termination, respectively. `_ssbegin()` is a wrapper function for `ssbegin` and `sstramp` ISA instructions. The pseudo code of `_ssbegin()` is shown in Figure 4.6.

`_ssbegin()` first saves the event handler's address and starts SENSE mode with an event handler trampoline using the `ssbegin` ISA instruction. The trampoline and the handler then follow the semantics defined by SENSE. `_sswatch()` is a wrapper function for the `sssub` ISA instruction to subscribe to data or instruction at address `ptr` for eviction event `cache_ev`. When an event occurs, the program control flow is transferred to the trampoline, followed by the preparation and the invocation of the handler. The CPU states are then restored before resuming the process. `_ssunwatch()` is a wrapper function for the `ssunsub` ISA instruction to unsubscribe the eviction event at address `ptr`. Finally, the `_ssend()` terminates the SENSE mode.

### 4.2.3 Action Module (AM)

The event handler first saves the interrupted context, including volatile registers and flag states, onto the enclave stack. Then, the handler reenters the paused SENSE mode and handles the event accordingly. Finally, the handler restores interrupted context and invokes `RET` to resume the interrupted program execution. An example is provided in Figure 4.6. We assume the event handler is a part of the trusted software component of the TEE.

SENSE provides default handlers, *i.e.*, `ABORT`, `INVARIANT`, and `THRESHOLD`, for common handling of the events (Figure 4.3).

**`ABORT`.** The `ABORT` handler simply aborts the enclave process upon the occurrence of the event, a pessimistic strategy adopted by various existing detection techniques [63, 64, 65, 66, 67, 51, 68, 49]. It provides the most constrained security policy for an enclave process that forbids any subscribed events. It targets events defined by the user that are absolutely forbidden from the normal execution of the enclave, such as writes to data from a malicious process during an atomic operation.

*Implementation.* Upon the eviction of members in the *watch set*, the `ABORT` handler will simply halt the enclave process by invoking `exit(0)`.

**`INVARIANT`.** The `INVARIANT` handler aims to preserve a safety property as an invariant for

```
void _sswatch(void *ptr, size_t size, ss_t type) {
  /* Add [ptr, size, type] tuple
      to the watch set if not found. */
  watch_set.add([ptr, size, type]);
  sssub(ptr, size, type);
}

void _ssunwatch(void *ptr) {
  /* Remove ptr in the watch set if found. */
  watch_set.remove(ptr);
  ssunsub(ptr, size, type);
}

void handler(int ss_info) {
  _ssbegin(handler);
  for (t in watch_set)
    _sswatch(t->ptr, t->size, t->type);
}

/* _ssbegin(handler) by TEE
      at enclave entry */
void enclave_program() {

  char mem[80];
  /* Subscribe to target memory. */
  _sswatch(mem, 80, cache_ev /* event type */);
  ...
  _ssunwatch(mem);

}
/* _ssend() by TEE
      at enclave termination */
```

**Figure 4.7:** An example of state-pinning model.

the enclave process. When the safety property is tampered with due to an SENSE event, the event handler could reestablish the property. Therefore, from the enclave perspective, the safety property is never broken. For example, a cache residency invariant for the enclave (*i.e.*, retain the specified cachelines in the cache) can be preserved by refetching the evicted cacheline in the event handler.

*Implementation.* To maintain the cache residency of the *watch set* during enclave operations, the `INVARIANT` handler will *re*-subscribe to all of the cache entries in the *watch set* so that the entries are prefetched again and stay in cache after eviction. Upon eviction on monitored cache entries under SENSE mode, the *watch set* is *re-fetched* by subscribing to all *watch set* members again, thus preserving the cache residency of the enclave process. From the application's perspective, the cachelines in the *watch set* are effectively pinned into the cache. A detailed example of the `INVARIANT` handler is shown in Figure 4.7. By calling the `_sswatch()` function with the security-critical data `mem` and an event type `cache_ev`, `mem` is put into the *watch set* for cache eviction events. Upon eviction on `mem` under SENSE mode, the *watch set* is *refetched* by calling the `_sswatch()` on all *watch set* members, thus preserving the cache residency of `mem`.

**THRESHOLD.** The `THRESHOLD` handler keeps a thread-local counter that is incremented each time the handler is invoked. The user could define a termination policy in terms of this counter, such as *terminate after n events are detected.* As there is not a universal threshold, the developer can tailor a custom threshold for each application. For example, a security policy for the AES T-table will likely differ from one for a data-intensive function, *e.g.*, a machine-learning algorithm.

*Implementation.* The threshold can be determined according to the *watch set* size and eviction times under normal circumstances. For instance, an AES-256 encryption key may be monitored under the *watch set* as four cachelines of 64 bytes. During encryption, if the key is evicted $k$ times under normal conditions, a user might set the threshold at $2k$, allowing for a small amount of noise while terminating upon detecting suspicious actions by invoking `exit(0)`. Recent studies [62] have proposed more precise methods for automatically determining the optimal threshold value by systematically modeling and quantifying information leakage.

**Custom event handlers.** Besides the default event handlers, SENSE allows developers to design and register custom event handlers tailored for other needs. However, they should follow the assumptions of proper enclave workloads (§4.1.1) as handlers handle events within the enclave.

### 4.2.4 Compatibility

**Backward compatibility.** Similar to supplementary processor features like Page Attribute Tables (PATs) and Memory Type Range Register (MTRR), SENSE offers on-demand cache event notifications while maintaining backward compatibility. SENSE effectively provides cache event notifications during execution only when processes are running under TEEs, which are indicated by the SENSE status bit of each cache entry and communicated with each cache request. Otherwise, from a perspective of software outside TEEs, a CPU with SENSE functions identically to one without SENSE. If the trusted hardware is not initialized and deployed for processes, SENSE is designed to not turn on the SENSE status bit by default to incoming cache requests, treating all executions as not under TEEs with cache event notifications disabled. Only when trusted hardware or firmware support is provided, do CPUs with SENSE differentiate for TEEs and offer its microarchitectural notification capabilities.

**Synergistic compatibility.** SENSE is compatible with all existing partitioning and randomization-based defenses against cache-based SCAs in TEEs [57, 58, 59, 60, 61], which attempt to address such SCAs at their root cause (*i.e.*, cache organization). This compatibility is due to the SENSE architecture-agnostic design, which does not rely on platform-specific hardware features and is independent of cache organizations (*e.g.*, set-associativity). Furthermore, when combined with existing partitioning-based techniques, SENSE can reduce additional hardware overhead to near zero. Existing partitioning and randomization-based solutions already extend cache entries with multiple bits (*e.g.*, 4 bits [57, 59]) for information tracking and verification, which can be shared with SENSE for tracking its status. Additionally, these solutions also extend cache controller logic [57, 58, 59, 60, 61] to enforce resource allocation and isolation, allowing SENSE's cache controller logic to be integrated with minimal additions.

**Architectural compatibility.** SENSE aims to offer ISA abstractions (Table 4.1) that are sufficiently generic to be implementable on diverse architectures (*e.g.*, `x86`, ARM, RISC-V, etc.) to enable microarchitectural event notifications. However, the microarchitectural implementation will likely vary across different chip designs. For example, the presence of a cacheline is generally easier to establish in an inclusive cache hierarchy than in a non-inclusive one.

### 4.2.5 SENSE for General Hardening

As SENSE provides notifications on finer-grained microarchitectural events, more advanced defense mechanisms are made possible with such capabilities.

*Case 1: On-demand loading of secure libraries under attack.* Software-based SCA mitigation techniques generally incur heavy performance overhead in order to hide secret-dependent microarchitectural traces with extra operations. For example, constant-time cryptographic algorithms can effectively defeat timing-based SCAs but incur performance overhead that largely hinders their adoption in applications. During normal executions, the performance overhead paid for constant-time cryptographic operations is unnecessary; on the other hand, with the concern of SCAs, it is insecure to rely on optimized but vulnerable libraries all the time. Ideally, applications want to benefit from the optimized performance in the less-secure libraries and pay the overhead of the secure libraries only when under attack, which is difficult to achieve.

SENSE can be utilized to reach this goal. The application can first execute using an optimized but potentially vulnerable library (*e.g.*, vulnerable OpenSSL). When malicious microarchitectural states are discovered under SENSE (*e.g.*, an excessive number of cache evictions), the more secure version of the library (*e.g.*, constant-time OpenSSL) can be loaded on demand by the event handler, replacing the potentially vulnerable one. We present the evaluation of such an application in §4.4.5.

*Case 2: Verifying contracts with the OS.* The OS is in charge of managing system resources with a higher privilege than applications. To this end, there is a hidden contract between applications and the OS: the OS will fulfill the requests from applications with due diligence and perform privileged tasks on behalf of applications faithfully. However, an untrusted OS does not always honor such contracts. To verify the contracts with the OS, applications can implement a verification logic in a trusted way using SENSE.

To illustrate, a concrete use case is whether the OS enforces cache coloring [187, 188] as it claims to be, *i.e.*, physical addresses that belong to a particular color are only accessible (*i.e.*, read, write, evict) by the threads assigned the same color. When using cache coloring to isolate cache accesses in SCA mitigations, one cache color can be assigned to at most one thread, thus preventing cache sharing between the victim thread and malicious threads. When a security-critical cacheline in the enclave application's *watch set* is evicted, the handler should therefore verify whether the cacheline is evicted by the enclave thread itself. If not, the OS is not following the contract and might be unfaithful. Further actions such as terminating the process can be enforced to protect the process from the malicious OS. We present the evaluation of such an application in §4.4.6.

## 4.3 Security Analysis

Each existing side-channel defense aims to mitigate a well-defined vulnerability and thus does not worry about increasing the attack surface. In contrast, SENSE, being a non-traditional interface that explicitly exposes microarchitectural information to TEEs, faces various unpredictable situations. We first introduce the security guarantees enforced by the SENSE architecture. Then, we analyze possible attack surfaces and show that SENSE does not empower attackers more than the system without it.

### 4.3.1 Security Guarantees

*1) Delivery of* SENSE *notifications is guaranteed.* The delivery of the notification is enforced by the trusted hardware (CPU). The SENSE *Notification Module* ensures that the control flow will be directed to the event handler trampoline when an event occurs under CPU SENSE mode. The trampoline in the enclave then guarantees the execution of the event handler.
*2)* SENSE *handlers are inaccessible from other threads.* As SENSE is core specific, SENSE handlers are only accessible by the enclave thread that sets the trampoline. When an event occurs or the SENSE thread is suspended, the SENSE mode pauses and the control flow is transferred to the SENSE trampoline (§4.2.2). The SENSE trampoline directs to the event

handler, and the address of the trampoline is registered in the SENSE control registers at the start of the enclave process; thus it is specific to the thread. Therefore, the event handler is only invokable by the enclave thread that registers the trampoline and no one else.

*3) Microarchitectural states are cleared when* SENSE *mode pauses/exits.* When an event occurs, the monitored microarchitectural states (*e.g.*, the watch set) are cleared (*e.g.*, flushed) before jumping to the trampoline to avoid leaving microarchitectural traces when events are not monitored, which is aligned with the strategy of existing TEEs.

*4) Security of event handlers.* SENSE guarantees only the execution of the userspace handler when the event occurs, while SENSE does not make any assumption about the security of the handler. Default handlers provided by SENSE (*i.e.*, `ABORT`, `INVARIANT` and `THRESHOLD`) can be used directly if they already meet the functionality requirement; however, SENSE does not verify whether custom handlers are secret-independent, audit the custom handlers for potential vulnerabilities, or prevent the custom handlers from intentionally leaking the secret, which should all be prevented when designing a new custom handler.

### 4.3.2 Attacker's Exploitation of SENSE

We examine whether the attacker can utilize SENSE to acquire extra information about the victim that is not available otherwise. We assume that there are two threads running, *i.e.*, one victim thread and one attacker thread. Within the TEE threat model, the victim thread typically operates inside a TEE, whereas the attacker can be any other thread outside of that TEE. Additionally, there's the potential risk of an attacker deploying a rogue TEE to subscribe to SENSE events, thereby monitoring the victim's behavior.

**Same logical core.** When running on the same core with the victim, the attacker cannot monitor events in the victim thread, as they cannot run at the same time. However, the attacker is able to learn about the absence of cache microarchitectural states, which is cleared after the SENSE process is context-switched out (*e.g.*, a timer interrupt). This does not leak information about the program activity since the cache microarchitectural states appear all as *cleared* (*e.g.*, all cache misses in cache) to the attacker.

**SMT and different physical cores.** When the attacker and the victim are located on different cores, they can interfere with each other through shared microarchitectural resources such as L1/L2 cache under SMT or the Last Level Cache (LLC) shared by different physical cores. There are three attack scenarios when SENSE is available to both the attacker and the victim.

*1) Only the victim is using* SENSE. That is, the attacker is not a TEE thread. The attacker can maliciously trigger an SENSE event (*e.g.*, evict the *watch set*) and invoke the corresponding handler registered by the victim. The attacker can also choose to invoke a specific handler that is less desired by the victim thread (*e.g.*, triggering abort instead of preserving an invariant) by targeting the corresponding enclave thread. The attacker can learn about the victim's *watch set*, which is the set of the memory objects in the enclave. It does not reveal useful information since the enclave memory is already managed by untrusted privileged software. The attacker cannot infer cache activities, as any cache probing will be notified to and handled by the enclave (*e.g.*, pin the cache).

*2) Only the attacker is using* SENSE. That is, the attacker is a malicious TEE thread targeting the OS and userspace. By requesting notifications on specific cacheline evictions, the attacker can infer the victim's cache access behavior with lower noise and delay, hence boosting the efficiency of an attack. More specifically, with SENSE, a malicious TEE thread can probe victim cache accesses without a traditional coarse-grained *timing measurement* phase, as SENSE promptly delivers notifications to the attacker. This reduces the overhead to launch an attack (as evaluated in §4.4.2). However, SENSE does not expose *new* information to a malicious TEE thread through notification delivery. The cacheline access information an attacker can subscribe to using SENSE can already be obtained without SENSE [7, 5, 6, 189].

Furthermore, an OS possesses alternate defense mechanisms to shield itself and its user processes from a potentially malicious TEE. For example, the OS can mitigate SMT attacks (*e.g.*, L1, L2 cache attacks) by leveraging Linux's core scheduling to ensure that threads belonging to a potentially malicious TEE are not co-scheduled on the same physical core with another process's threads [51, 50]. Cache partitioning strategies [57, 58, 59, 60, 61] (or coloring techniques [190, 191, 192]) can be employed to combat cross-core attacks (*e.g.*, LLC). By designating a specific partition, an OS can isolate a potentially malicious TEE, and this approach can seamlessly integrate with SENSE as discussed in §4.2.4.

*3) The attacker and the victim are both using* SENSE. That is, both the attacker and the victim are TEE processes. In this case, the attacker has extra capability due to simultaneous handling of the event. We give an example security caveat under such a circumstance. Suppose that both the attacker and the victim monitor a shared data region (*e.g.*, AES T-table), each with an SENSE handler registered. Two of the cachelines monitored by the victim, $A$ and $B$, map to the same cacheline in the LLC. The victim's `THRESHOLD` handler checks the cache eviction events and aborts the process if it exceeds a preset threshold, while the attacker's `INVARIANT` handler waits and refetches the evicted entry. Upon cache eviction, the attacker and the victim will thus both get notified simultaneously, followed by the invocation of corresponding handlers.

Suppose that the victim program touches $A$ and $B$ in a sequence, and originally $B$ is in the LLC. The initial threshold counter ($tc$) of the victim is $0$. When the victim visits $A$, $B$ is evicted from the LLC and replaced by $A$, which triggers both handlers. The victim handler increases the counter ($tc = 1$); the attacker handler refetches $B$ into the LLC. When the victim visits $B$, since $B$ is already in the LLC, no eviction is needed. However, if there is no attacker, $B$ will not be present in the LLC; thus, one more eviction is needed to replace $A$ with $B$ ($tc = 2$). In this case, the attacker earns one eviction quota, which can be used to perform malicious evictions (*e.g.*, *prime* the eviction set) while avoiding detection by the victim handler.

The root cause of such a problem lies in a lack of enclave identity when monitoring cache entries. As a result, the notification cannot be sent to only the monitoring thread but to all enclave threads under SENSE. This causes the unpredictable interaction and synchronization among handlers from different sources. Such a hidden interaction of cache microarchitectural events among different handlers is difficult to prevent, as both the method

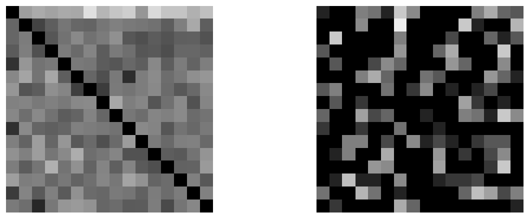

**Figure 4.8:** Color matrices showing cache hit patterns of the first AES T-Table ($k_0 = 0x00$) under Prime+Probe attack. Left: attack without SENSE; Right: attack with SENSE cache `INVARIANT` handler.

of interaction and the number of interacting parties are unknown.

This unpredictable interference among handlers can be mitigated by integrating the enclave identity into the monitored cache entries so that the event notifications can be demultiplexed according to the enclave identity. However, this approach increases the hardware overhead, while using such an interference to collect meaningful microarchitectural traces is only theoretically possible. Nonetheless, the hardware overhead of tracking enclave identities of the monitored cachelines can be eliminated if SENSE is deployed together with other partitioning-based mitigation techniques for TEEs that are already tracking the enclave identities at the cacheline level ([57, 60]). When SENSE is deployed alone on vanilla processors instead, to minimize the likelihood of such a threat, it should be configured with the *exclusive monitoring* feature (§4.2.2) offered by the SENSE architecture while sacrificing the functionality of *event broadcasting*.

## 4.4 Evaluation

**Experiment setup.** We implement and perform our evaluation on SENSE using a cycle-accurate gem5-based [111] `x86` emulator with its out-of-order (O3) CPU model at 4 GHz and the default Classic Memory System. The server for our gem5 emulation is running Linux kernel version 4.8.1 and is equipped with a 4-core Intel i7-6700K CPU (Skylake) operating at 4 GHz, and 64 GiB of RAM. Each evaluation experiment conducted is run 10 times and we calculate the average values as the final results.

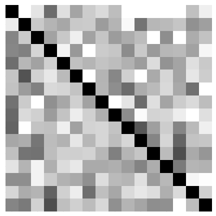

**Figure 4.9:** Cache hit patterns using SENSE notifications for probing. The signal is stronger compared with Figure 4.8 Left.

### 4.4.1 Security Evaluation

**Harden AES T-Tables.** To evaluate SENSE on closing timing-based cache SCAs, we showcase the Prime+Probe attack against the the vulnerable AES T-table implementation with and without the protection of SENSE. Although the T-table is disabled by default in OpenSSL, it is still widely used to evaluate new and existing SCAs. The attack implementation is based on strategies from previous works [193]. During encryption, the T-table entries are used together with the secret key $k$ to compute the ciphertext from the plaintext $p$. Specifically, the key bytes and plaintext bytes are used as indexes to access the T-table entry $T_j\left[p_i \oplus k_i\right]$, where $i \equiv j \mod 4$. An attacker is able to use Prime+Probe to leak the access patterns of the T-table entries, which reveals the values of $p_i \oplus k_i$. As a result, $k_i$ can be computed when $p_i$ is chosen by the attacker.

Each cacheline (*i.e.*, 64B) can hold 16 T-table entries. We launch the attacks against the first line of the first T-table `Te0` (*i.e.*$k_0 = 0x00$) to leak its access patterns for demonstration [139]. First, we run the attacks without SENSE protection. Then, we monitor cache eviction events of the encryption operation using SENSE, assuming the encryption operation is typically the critical section executing under TEEs. The `INVARIANT` handler refetches the evicted cacheline, as described in §4.2.3, to effectively pin it in the cache throughout the attack so that the timing side channel is concealed from the attacker. We show the sampled cache access patterns of the first line in T-table `Te0` in Figure 4.8. The T-table without

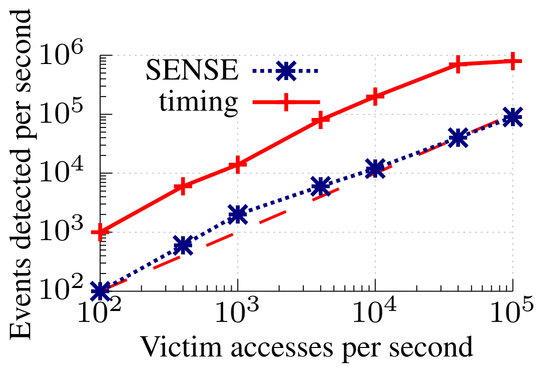

**Figure 4.10:** Cache access detection rates using the timing channel and using SENSE notifications. The $y = x$ line indicates perfect performance, *i.e.*, all events are detected without false positives. Being more congruent to the $y = x$ line means lower false positive rate.

SENSE protection reveals a clear cache hit pattern under the Prime+Probe attack, while the T-table protected by SENSE does not leave useful microarchitectural traces for the attacker to infer secret key bits.

### 4.4.2 Attacker's Exploitation of SENSE

We evaluate the efficiency benefits brought by SENSE to a malicious TEE (§4.3.2). On average, identifying *a single cache access* via SENSE notifications is ~8× faster than probing using a traditional timing channel (*i.e.*, 576000 vs. 4649000 emulated CPU cycles, respectively), allowing the attacker to carry out more probing attempts within the same timeframe. Figure 4.10 represents the success rates of detecting victim memory accesses through both techniques. For this experiment, the victim consistently accessed a single memory location while the attacker aimed to monitor these accesses. The results highlight that while both methods can identify victim accesses, SENSE notifications have a reduced false positive rate compared to the conventional timing channel. This improved accuracy stems from SENSE's immediate hardware callback upon cache evictions, obviating any requirement for a potentially error-prone *timing measurement* phase as in traditional cache attacks.

We further show the efficiency benefits during an actual attack using the same Prime+Probe attack outlined in §4.4.1. The primary segment of Prime+Probe utilizing SENSE mirrors the *prime* phase of a regular Prime+Probe attack, except that it primes by subscribing the eviction set to cache eviction events first. Once the *prime* phase is completed, the attacker simply waits for a SENSE notification, from which the attacker can conclude that another program has accessed an address in the target cache set. This is similar to the information leaked by a regular Prime+Probe attack. As shown in Figure 4.9, assisted by both the faster execution speed and the lower false positive rate of SENSE notifications, the signal is stronger when using SENSE for Prime+Probe.

### 4.4.3 Performance Evaluation

We first evaluate the performance of each SENSE module, *i.e.*, *Subscription Module (SM)*, *Notification Module (NM)*, and *Action Module (AM)*, using benchmark PolyBenchC ([172]). Next, we evaluate the overall performance of SENSE when protecting AES encryption from attack.

**Benchmark performance of SENSE without attacks present.** The overhead of SM comes from prefetching the secret data into cache and marking cache entries as SENSE monitored. The overhead from NM lies in the extra CPU control logic that relays the cache event information, microarchitectural state cleaning upon notification, and control transfer to the trampoline. The overhead of AM solely depends on the handling logic.

We use PolyBenchC [172] to conduct the performance evaluation. Each benchmark in PolyBenchC has a small matrix function, which we treat as the minimal critical section (§4.1.1) executing inside a TEE and thus monitored by SENSE, in our experiment. The functions behave similarly to how look-up tables are used in AES encryption. We then launch the programs and measure the execution time of each module by the CPU cycles emulated in gem5.

*1) Performance of Subscription Module (SM).* The SM prefetches all security-critical memory objects by iterating through the objects as cacheline-size data blocks (*i.e.*, 64B) to be put under SENSE monitoring. The results are shown in Figure 4.11. On average, the SM incurs a 1.2% overhead compared to the execution time of the original program, which is negligible.

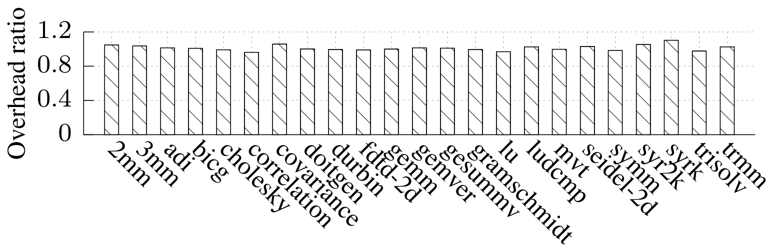

**Figure 4.11:** Performance overhead of the Subscription Module (SM). Average overhead is 1.2% of the overall execution.

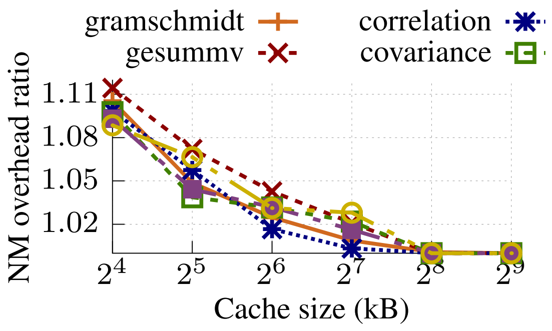

**Figure 4.12:** Performance overhead of the Notification Module (NM).

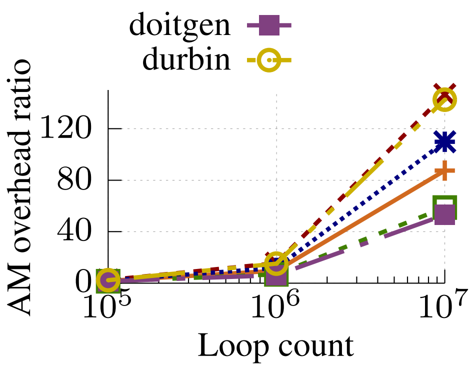

**Figure 4.13:** Performance overhead of the Action Module (AM). The cache size used is 32kB to incur more evictions.

*2) Performance of Notification Module (NM).* The result is shown in Figure 4.12. Under normal cases, as the monitored data size is smaller than the cache size, cache evictions on the monitored data do not occur often, and thus the performance impact is negligible. Up to a cache size of 256kB, the overhead incurred by NM on PolyBench kernel functions is imperceptible. As a showcase of extreme scenarios, we also radically reduce the size of cache of the gem5 emulation configuration to as low as 16kB to intentionally increase cache contention so that more events can be triggered. As shown, the smaller the size of cache, the more performance overhead NM incurs, as the cache evictions due to cache conflicts occur more often under a smaller cache, which matches our initial speculation. In conclusion, with a realistic cache size and a minimal enclave program, the NM incurs negligible performance overhead, while under extreme cases where the cache size is impractically small to serve the enclave program, extra overhead will be caused by increased notifications.

*3) Performance of Action Module (AM).* To analyze the performance overhead of AM, we use dummy handlers that perform busy waiting inside a loop. By increasing the size of the loop variable, we can simulate the increase of handler complexity and analyze the correlation between performance overhead and complexity of the handler. To make the behaviors of the handler more observable, we use a cache size of 32 KB, which is shown to trigger the event quite often in the previous experiment. The result is shown in Figure 4.13. Not surprisingly, as the handler becomes more complex, the AM incurs more performance overhead. Overall, the performance of the handler dominates the overhead of SENSE. Note that this is only to showcase the overhead of the AM under extreme cases with particularly small cache size and over-complicated handler logic. With practical cache size and minimal enclave programs, events are rarely observed, as shown in the previous experiment, and the complexity of default handlers is much simpler than the ones in the experiment. Therefore, the AM also incurs negligible performance overhead under practical use cases without attacks in place.

**Performance of SENSE under attack.** We compare the performance impact of SENSE when protecting AES encryption under attack by using the `INVARIANT` handler. The results are shown in Figure 4.14, where the red dotted line indicates the baseline performance of AES encryption without attack. The AES encryption is typically fast without inducing self-evictions. When under attack, T-table entries are intentionally evicted from the cache (*e.g.*, using *prime* phase) for launching cache SCAs such as Prime+Probe. We observe a minor and negligible decrease of performance when experiencing (potentially malicious) cache evictions on T-table entries during AES encryption without SENSE. When protected by SENSE, the performance of AES encryption quickly decreases as the number of experienced evic-

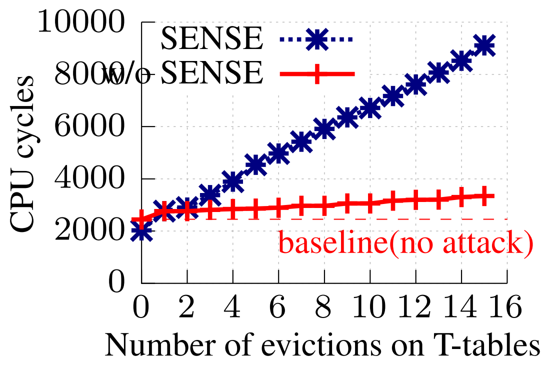

**Figure 4.14:** Performance of AES encryption w/ and w/o SENSE under attack. The red dotted line indicates the baseline performance of AES encryption without experiencing attacks.

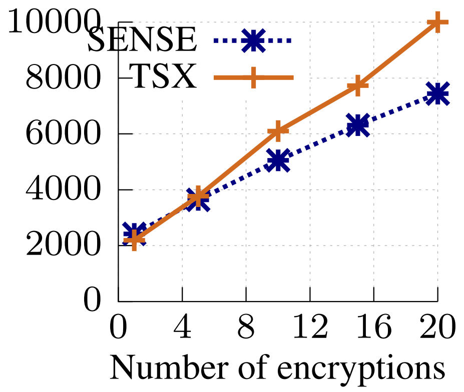

**Figure 4.15:** Performance of AES encryption under SENSE and TSX *without attack*.

tions increases. The performance degradation is mainly caused by the SENSE notifications and the event handling by the `INVARIANT` handler to fetch the evicted T-table entries back into the cache. As a reference, the Intel TSX-based mitigation technique [48] against cache SCAs is reported to reduce the performance of AES encryption with T-table to 23.7% ($\sim$4.22$\times$) of the baseline performance under a Prime+Probe attack. The lower performance while experiencing evictions is justified given that the timing channel is eliminated. Besides, the attacker has to heavily load the whole CPU to launch the attack, which occupies a significant amount of hardware resources and largely slows down the performance already [48].

### 4.4.4 Comparing with TSX

SENSE is influenced in part by Intel TSX [194], which was deprecated recently [147, 148]. TSX has been used to address SCAs [68, 49, 48] with noteworthy caveats. The developer will need to partition the program into sections that are small enough to fit within a TSX transaction, which is inflexible and difficult to achieve. Additionally, each TSX transaction must be restarted from the beginning after a transactional abort, while executions can be resumed at the point where the event was detected under SENSE. Furthermore, TSX is only available on Intel platforms and was not originally motivated by SCAs, thus not applicable for monitoring microarchitectural events other than cache evictions, while SENSE, being agnostic to processor architecture and microarchitecture, can be easily extended to other cache events or events of other microarchitectural components.

Despite the limitations of TSX, we consider TSX as an existing technique to compare with SENSE on mitigating cache SCAs. We conduct rounds of AES encryption both under SENSE protection and within TSX transactions and compare the performance overhead. The results are shown in Figure 4.15. Due to the higher chance of evictions induced by more encryptions, more TSX aborts and SENSE notifications are triggered, thus the decreased performance. However, the performance of SENSE decreases more slowly than that of TSX. One reason is that instead of retrying from the beginning after a TSX abort, SENSE handles the event *in place* and restores the operation at the interrupted location. This makes SENSE more scalable than TSX to mitigate cache-based SCAs.

### 4.4.5 On-demand Loading of Cryptographic Functions

We demonstrate that upon a notification, the event handler is able to replace the functions that are optimized but vulnerable to SCAs with the secure versions for subsequent operations while only paying a small overhead to perform the switching. We first collect the execution time of two OpenSSL AES encryption implementations (*i.e.*, with and without T-table), as well as the overhead of replacing the vulnerable one with the one without the T-table, which is considered secure. As shown in Table 4.2, compared to the execution time of the library functions, the overhead of function switching is negligible.

The performance of AES encryption with the T-table protected by SENSE becomes

**Table 4.2:** Performance impact of replacing cryptographic functions. Values are in CPU cycles. The switching overhead is around 1.46% compared to encryption operations, and is only paid once.

| AES T-table | Function switch | AES no T-table |
|---|---|---|
| 2443 | 76 | 5195 |

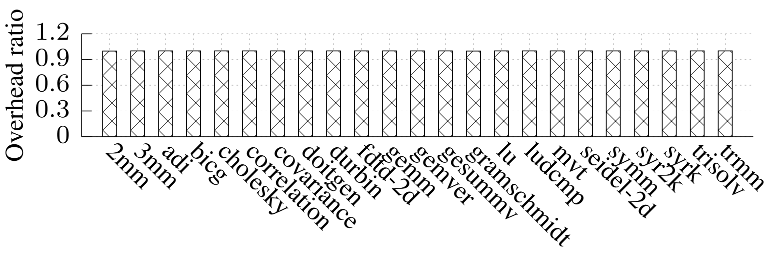

**Figure 4.16:** Performance overhead of the Action Module (AM), when verifying cache coloring. Cache size is 32kB to incur more events.

comparable to that of AES encryption without the T-table (*i.e.*, 5195 cycles) when experiencing about seven malicious evictions on T-table entries, extrapolated in Figure 4.14. Therefore, to maintain good security and performance levels, the event handler can perform the function switch when observing more than seven evictions on the protected entries, thus only paying the performance overhead of the secure version when handling events becomes too expensive.

### 4.4.6 Verification of cache coloring

We enable the event handler that verifies cache coloring on PolyBenchC programs as well to analyze the performance overhead of such a verification. The identity of the thread evicting the monitored cacheline is supplied to the enclave stack by the CPU. Then, the handler reads the corresponding data from the stack and checks whether the thread that caused the cache eviction matches its own thread identity. Similar to previous experiments, we use a cache size of 32kB to make the behaviors of the handler more observable. The performance overhead of cache color verification is negligible, as shown in Figure 4.16.

### 4.4.7 Hardware and Memory Overhead

We implemented the RTL [195] for SENSE and evaluated its hardware overhead for cache events, following previous works [57, 60]. The hardware logic overhead mainly stems from the SENSE cache controller logic in SM and the SENSE Fault interrupt controller logic in NM. Compared to an 8-core Xeon Nehalem [196] of 2,300,000,000 transistors, the logic overhead of SENSE is minor, estimated at 0.1%. The logic overhead can be further reduced if the target platform support updating the interrupt controller by firmware or microcode, which do not incur hardware modifications. The memory overhead includes the additional register components per logical core and two bits per cache entry for tracking SENSE status. When targeted platforms possess unused reserved registers and register bits, which is a common case in modern processors, the SENSE register hardware overhead can be eliminated. When combined with existing isolation-based techniques, SENSE can significantly reduce additional hardware overhead even further, as discussed in §4.2.4.

## 4.5 Discussion

**Limitations.** First, the size of the monitored data is bounded by the size of the underlying microarchitecture. For example, if the size of data monitored for cache eviction events exceeds the size of the cache, prefetching the cachelines under SENSE will always evict another cacheline, causing a deadlock. Therefore, knowledge of the target platform is needed when deploying SENSE applications. Second, interaction between SENSE blocks and code outside SENSE blocks may introduce side effects. For example, SENSE will not track the memory event when the monitored memory is copied into another memory region. That is, SENSE applies only to the original instance of the memory in the microarchitecture. Taint-based approaches might be adopted to propagate the monitoring status outside SENSE blocks.
**Defending other SCAs.** SENSE is most practical for side channels that are stateful, such as TLBs and caches (in this thesis), which are the most widely researched side channels.

SENSE could also be adapted to detect controlled-channel (*i.e.*, architectural) SCAs such as A/D-bit assists [120] and page faults [25]. There is currently no way for an SGX enclave, for example, to detect when an A-bit assist has occurred. We leave those events for future work. We admit that port contention [197, 198], on the other hand, is challenging for SENSE to mitigate as it occurs at very high frequency, and thus the event notifications might happen so frequently that software cannot make forward progress.

**Updating SENSE.** To update SENSE, developers only need to extend the Subscription Module (SM) for new microarchitectural events and handler designs, as the Notification Module (NM) is not specific to microarchitectural components and all notifications are reprsented as SENSE Faults. Developers should first identify event paths in microarchitectural components and incorporate them into the SM. For instance, adding a speculative execution event requires including branch predictors and caches. Developers must then insert a SENSE status marker in the component to indicate SENSE monitoring and ensure correct notifications. This may involve extending branch predictors and caches with an extra bit for monitoring status. Developers should also follow event paths to implement monitoring status propagation and define detailed event information to create effective handlers in userspace TEEs. For example, providing handlers with branch prediction history and prediction results can help users detect signs of malicious branch predictor training.

## 4.6 Related Work

We categorize cache side-channel defenses into two broad classes: 1) isolation-based and 2) detection-based. In this section, we focus only on the most relevant works to SENSE.

### 4.6.1 Isolation-based Defenses

**Partition-based defenses.** The partitioning-based defenses propose new cache architectures that allocate cache resources (cachelines or ways) exclusively to protected domains (*e.g.*, TEEs) [177, 61, 199, 200, 58, 59, 60]. These defenses can result in cache under-utilization when assigned cache ways are not evenly used by a protected domain, as the unused cachelines are blocked for all other domains on the system. Other approaches [201, 57] are more flexible, as they partition the cache on a cacheline basis. However, they still do not provide strong security guarantees against occupancy-based attacks, as they do not enforce strict partitioning. In memory page-coloring schemes [190, 145, 202, 176], the mapping from physical memory addresses to cachelines ensures that cachelines used by sensitive applications do not overlap. One issue with page-coloring is its reliance on OS or hypervisor, which are untrusted under TEEs. Furthermore, the assignment of cachelines is static. SENSE, on the other hand, offers flexible monitoring on exact cache entries that is satisfied upon TEE initialization at runtime. SENSE bypasses the untrusted privileged software as a native interface and does not affect the memory layout.

**Randomization-based defenses.** Randomization-based defenses employs randomized mapping tables to randomize the mapping of memory lines [201, 203, 204]. Cryptographic randomization techniques [205, 206, 207, 208, 209] aim to circumvent the storage overhead of large randomized mapping tables by depending on cryptographic primitives to consistently generate randomized mappings. These methods can only diminish the bandwidth of cache attacks rather than completely eradicate them. Attackers can still execute eviction operations when they access a sufficient number of lines across a vast array of cache sets. Furthermore, some strategies may experience significant performance decline when implemented on the considerably larger last-level cache. SENSE, on the other hand, fundamentally eliminates the aforementioned unreliability and inflexibility by providing timely, accurate, and flexible notifications directly to userspace TEEs.

### 4.6.2 Detection-based Defenses

Detection-based defenses can be primarily divided into two categories: signature-based [210, 211, 212, 213, 64, 140] and anomaly-based [214, 215, 216]. Some methods [63, 217, 67] utilize a combination of both signature- and anomaly-based detection techniques.

**Signature-based.** Demme *et al.* [210] employ L1 hits events in Hardware Performance Counters (HPCs) to detect malware and cache SCAs. Allaf *et al.* [211] suggest another signature-based detection technique to identify Prime+Probe and Flush+Reload attacks using machine learning (ML) models and HPCs while running an AES cryptosystem. The hardware events utilized include core cycles, reference cycles, and core instructions. NIGHTs-WATCH [212] can detect access-driven cache SCAs using various ML models that leverage LLC misses and CPU cycles in HPCs. This method trains the model under specific system loads, but it is unclear whether it performs well under unknown system loads. Mushtaq *et al.* [213] apply linear and non-linear ML classifiers to detect Prime+Probe attack variants running under the AES cryptosystem. HexPADS [64] uses cache miss rates and page fault values to detect Flush+Reload and cache template attacks [140].

**Anomaly-based.** CacheShield [214] is an anomaly-based detection mechanism for legacy software (victim application) that monitors LLC cache misses using HPCs. Bazm et al. [215] detect cross-VM cache SCAs by utilizing hardware fine-grained information provided by Intel Cache Monitoring Technology (CMT) and HPCs, following the Gaussian anomaly detection method. SpyDetector [216] can identify Flush+Reload, Flush+Flush, and Prime+Probe attacks running on RSA, AES, and ECDSA cryptosystems by monitoring L3 cache and L1 data cache through HPCs.

**Signature and anomaly-based.** Chiappetta et al. [63] propose a machine learning-based detection mechanism for Flush+Reload attacks on AES and ECDSA cryptosystems, monitoring L3 access to detect attacks. CloudRadar [67] is a signature and anomaly-based detection system that detects attacks in two steps. The first step involves identifying cryptographic applications using branch instructions and dynamic time warping. The second step defines a criterion for distinguishing between benign and malicious programs. CloudRadar considers an attack to have occurred when the detected value exceeds this criterion. Alam et al. [217] present a multi-layer detection approach based on machine learning, which collects microarchitecture events (e.g., branch misses, LLC accesses, and LLC misses) during attacks. They train machine learning models based on these events to detect attacks.

The aforementioned works rely on microarchitectural events as feature vectors for detection. However, due to the limited capacity of microarchitecture components, they are highly susceptible to interference from system loads. These heuristic approaches lack robustness and tend to experience high false positives and false negatives. In contrast, SENSE detects potentially malicious behaviors at their exact locations and promptly notifies userspace TEEs, preventing the delayed awareness of potential attacks experienced in signature and anomaly-based strategies. Furthermore, SENSE enables flexible responses to suspicious behaviors, thanks to the rich microarchitectural information it provides.

# CHAPTER 5
# PORTAL: FAST AND SECURE DEVICE ACCESS WITH ARM CCA FOR MODERN ARM MOBILE SYSTEM-ON-CHIPS (SOCS)

## 5.1 Overview

### 5.1.1 Motivation

As more devices are integrated into Arm SoCs, managing secure environments becomes increasingly complex. Including many devices in a single Arm SoC presents challenges for Arm CCA's practicality upon its upcoming release to mobile Arm SoCs.

**Limitations of existing approaches.** Specifically, existing solutions for secure device I/O in Arm CCA do not align with the ongoing architectural evolution of Arm SoCs for several reasons:

*1) Performance and TCB overhead:* Existing shared memory solutions [80, 43, 44] rely on software-based memory encryption which causes significant performance overhead (more than 40%), defeating the original goals of using devices such as accelerators, while recent study [79] requires non-trivial modification on hypervisor hardware and significantly bloats the TCB. Worse, memory encryption naturally hinders existing mature performance optimizations such as memory deduplication [76], opposing mobile platforms that demand real-time processing.

*2) Inscalability:* The performance overhead increases when multiple peripheral devices access unified memory simultaneously [30, 31, 32, 33, 34, 35, 36]. Each device must encrypt every memory access and undergoes complex key management overhead for multiple sessions. Current solutions do not scale with the increasing integration of devices in Arm SoCs, hindering the adoption of Arm CCA in the most prevelant mobile processor market.

*3) Inflexibility:* In order to enforce exclusive access, existing solutions employ one-to-one binding [77, 78, 79, 80] between the VM identity and the target device, which is persist throughout the VM's lifespan. Such a direct binding hinders the support of multiple peripherals or dynamic attachment of peripherals.

*4) Power inefficiency:* Mobile Arm platforms operate under stringent power efficiency requirements. The computational overhead associated with frequent memory encryption for various peripheral devices increases energy consumption substantially [81, 82, 83, 84, 85, 86, 87], making Arm CCA even less appealing for mobile Arm platforms.

*5) Lack of device generality:* Constrained devices that lack the capability to support cryptographic operations, including key negotiation and secure encryption, cannot be integrated into existing solutions that demand point-to-point encryption.

**Reevaluating memory encryption in Arm SoCs.** Historically, Memory encryption arose to secure data in transit and at rest within complex processor and memory architectures. This was crucial in shared and cloud computing environments to protect data from unauthorized access at the hardware level. Given the strong physical security and low risk of physical attacks on Arm SoCs, the need for memory encryption in Arm CCA for mobile processors should be reconsidered. In environments where processors are not physically threatened, strict memory access controls (*e.g.*, hardware-based GPC in Arm CCA) may provide sufficient protection without the drawbacks of memory encryption. We believe that carefully redesigning Arm CCA memory isolation can ensure data security without memory encryption, maintaining performance, scalability, and power efficiency for mobile Arm SoCs.

### 5.1.2 Targeted Platforms

We consider SoC-based Arm processors where the memory is embedded on chip (*i.e.*, integrated memory) and is shared between the CPU and peripheral devices (*i.e.*, unified memory model), a common configuration for mobile Arm processors. We assume that communication within the Arm processor package is secure and physical attacks are infeasible, owing to its built-in resilience against tampering and interference with interconnections (§2.5). As a result, the data in transit among the memory, the processors, and peripherals on chip, remains secure against physical memory attacks.

### 5.1.3 Threat Model

We assume that the next-generation Arm SoC will incorporate security primitives including the RME, CCA, and a hardware root of trust to support secure boot and remote attestation. We trust the Realm and the Root world that hosts the RMM and the Monitor, conforming to the TCB of Arm CCA. We assume a strong attacker who controls both the *Normal* world and the *Secure* world, including the host OS and the VMM. The attacker aims to leak or manipulate the sensitive data transmitted between the confidential VM and the peripheral device. The attacker can access the unified memory holding the sensitive data or employ DMA-capable peripherals to execute similar attacks. Additionally, the attacker might disrupt the isolated execution environment by compromising memory management and modifying the states of device registers. The attacker could also introduce malicious device tasks to access or tamper with sensitive data within other realms. We assume that all platform devices integrated into the SoC packages are not forgeable.

**Scope.** PORTAL targets the mobile Arm SoCs with integrated memory and devices, and assumes that it is infeasible to physically tamper with the processor package and probe the internal structures to leak data. We do not consider denial-of-service attacks from the malicious hypervisor or host OS. We also do not consider speculation attacks and side-channel leakage due to microarchitectural implementation.

### 5.1.4 PORTAL Overview

The core of PORTAL is a strictly isolated plaintext shared memory region, called PORTAL region, for data transmission between Arm CCA confidential VMs (*i.e.*, Realm VMs) and integrated devices on Arm SoCs. PORTAL ensures that only dedicated Realm VMs and

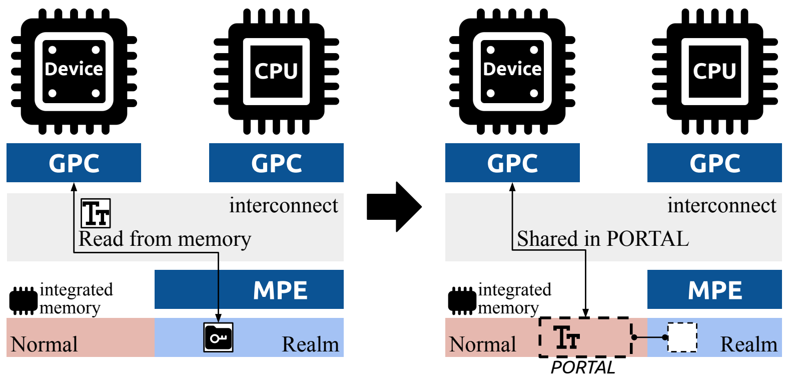

**Figure 5.1:** Access modes of integrated devices. Left: encrypted memory in native Arm CCA. Right: plaintext memory access through PORTAL.

peripherals can access the PORTAL region, while unauthorized entities, including privileged software (*i.e.*, VMM and host OS), are prohibited from access. PORTAL removes the requirement for memory encryption against physical attacks by relying on the robustness of Arm SoC packages.

Specifically, when authorized confidential VMs and peripherals initiate memory transactions to the PORTAL region, PORTAL configures their GPTs so that the PORTAL region is treated as the Normal world PAS, allowing them to issue memory transactions without encryption. The CPU and SMMU stage-2 translation tables are utilized to ensure Realm VM-level and device-level isolation. In contrast, the PORTAL region is configured as the Root world PAS for unauthorized entities, blocking their accesses.

**Challenges.** The design of PORTAL faces three key challenges. **C1:** The current SMMU design lacks Realm VM context (*i.e.*, whether a device belongs to a Realm VM) and cannot issue memory transactions targeting the Realm PAS. Meanwhile, the SMMU hosts multiple page tables in system memory (*i.e.*, *Normal world* and *Secure world*), which is considered untrusted under Arm CCA. PORTAL thus establishes an isolated memory region in the Normal world PAS for device communication, while securing it by restricting access to authorized entities. **C2:** It is critical to ensure PORTAL does not bloat the TCB of trusted components of Arm CCA. PORTAL uses a dedicated Realm VM, the *System Realm*, for critical management tasks (*e.g.*, configuring the SMMU), preserving the original TCB of EL2 and EL3 without expanding the RMM in EL2 or the Monitor in EL3. **C3:** As PORTAL removes memory encryption and adopts a plaintext-based shared memory region for device communication, the strict isolation enforced by PORTAL must span all layers of the Arm CCA stack to maintain data integrity and confidentiality. We conduct an in-depth security analysis on PORTAL, and demonstrate that PORTAL does not open new attack surfaces and the design decisions are well justified (§5.4.2).

## 5.2 Design

### 5.2.1 Protected Memory Regions

**PORTAL region.** PORTAL strictly isolates a shared memory region, called PORTAL *region*, for a particular pair of authorized Realm VM and device to achieve secure I/O without encryption. However, the existing SMMU under CCA is not aware of the Realm world context and all devices are treated in the *Normal* world. Therefore, a PORTAL region must reside in the Normal world to be accessible to both the Realm VM and the device. PORTAL utilize the stage-2 translation tables and SMMU translation tables to enforce memory isolation. The stage-2 translation table for Realm VMs is managed by the trusted RMM and is used to isolate mutually distrusting Realm VMs. PORTAL uses the RMM's stage-2 translations table to ensure that when a PORTAL region is mapped to a particular Realm VM, it will not be accessible by other Realm VMs that do not possess the mapping. As a result, even though GPC allow any Realm VM to access a PORTAL region, which is within the Normal world PAS, RMM's stage-2 translation table ensures the isolation of the PORTAL region from unauthorized Realm VMs. Similarly, PORTAL configures the SMMU translation table to ensure that only the authorized device has the mapping to the specific PORTAL region in its stage-2 translation. However, as the SMMU is managed by the untrusted host OS and the VMM, it can be manipulated by the attackers to allow attacker-controlled devices to access unauthorized PORTAL regions, demanding effective protection measurements. We discuss how PORTAL protects the SMMU in Section §5.2.2.

### 5.2.2 Protection of the SMMU

**Protecting the SMMU data structures.** Hosting the SMMU data structures (*i.e.*, Stream Tables and stage-2 translation tables per device) in highly privileged software layers such as the trusted Monitor or the RMM can prevent access from the attackers. However, the SMMU, without the corresponding permission, cannot perform accesses to the SMMU data structures located in either the RMM or the Monitor. Therefore, PORTAL hosts the SMMU data structures in a PORTAL region, which resides in the Normal world PAS and can be freely accessed by the SMMU, while isolated from the host OS and the VMM to prevent malicious manipulations.

**Protecting SMMU management.** In addition to protecting the SMMU data structures, the code logic that manages the SMMU should be secured as well. Specifically, originally located in the untrusted host OS and VMM, the SMMU management code should be relocated to memory regions inaccessible by the attackers. Existing approaches [80, 79] choose to move the SMMU management code to the trusted Monitor or the RMM to achieve intrinsic security offered by Arm CCA. However, as the Monitor and the RMM are the most security-critical layers of Arm CCA, introducing the SMMU management code will enlarge the TCB and might cause total compromise of Arm CCA (*i.e.*, both the Root and the Realm world) if security flaws exist in the added logic. In addition, as the RMM and the Monitor are located in the high privileged exception levels, it incurs additional overhead of context switching and TLB/cache flushing whenever the Realm VM, the host OS, or the VMM request SMMU updates.

PORTAL employs a dedicated Realm VM called *System Realm* to manage the SMMU on behalf of the host OS and the Realm VM that is authorized to communicate with the device. The System Realm owns the PORTAL region that hosts the SMMU data structures. The System Realm also has exclusive access to the MMIO registers that configure the

SMMU and its data structures, as these registers can be used by the attackers to compromise the security of PORTAL (*e.g.*, turn off stage-2 translations or the SMMU entirely). This is achieved by mapping the MMIO configuration registers only to the System Realm's translation table, forcing every SMMU-related operation to be routed to and handled by the System Realm. Note that the Stream Table I/O page tables are set up using the memory pages that belong to the System Realm, and are inherently only accessible by the System Realm itself, the RMM, and the Monitor.

We assume that the System Realm is implemented and distributed by trusted vendors such as Arm or SoC vendors. Since the System Realm is also an instance of Realm VM, it benefits from the existing Arm CCA security guarantees. First, based on the attestation report provided by Arm CCA, the Realm VM wishing to access the PORTAL region can verify whether the System Realm is provisioned by the trusted vendors. Meanwhile, the measurement provided in the attestation report validates whether the expected code logic (*e.g.*, no intentional information leak) is running in the System Realm. In addition, potential vulnerabilities in the System Realm (EL1) will not compromise the RMM (EL2) and the Monitor (EL3), unlike the existing approaches. Lastly, to reduce the overhead caused by frequent world switching and exception level changes, PORTAL provides an exclusive interface to interact with the System Realm. Through this interface, Realm VMs are able to send commands to the System Realm to configure the SMMU on demand. We provide more details about the PORTAL System Realm interface in Section §5.2.4.

### 5.2.3 Memory Isolation

All memory requests generated by peripheral devices are checked by the SMMU for access control and address translations provided in the I/O page table. As PORTAL regions belong to the Normal world PAS, memory transactions from devices to a PORTAL region are marked as Normal world transactions. However, since multiple devices can be attached to the same SMMU, a device can access any existing PORTAL region using Normal world

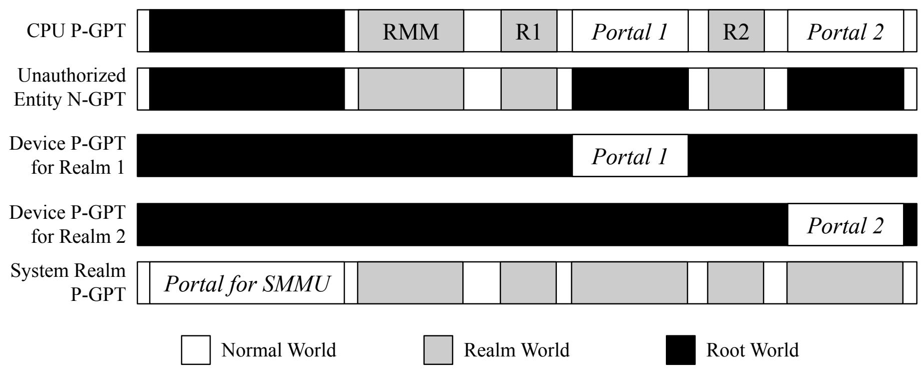

**Figure 5.2:** Memory accessibility for authorized and unauthorized entities in different security states (*i.e*., Realm or Normal) based on GPT configurations.

memory transactions, even when the particular PORTAL region is established for other devices. PORTAL thus requires device-level isolation within the Normal world to achieve two-way isolation between a dedicated pair of Realm VM and device, which can be achieved through the existing access control mechanism provided by the SMMU.

**Intra-realm and device-level isolation.** Arm CCA GPC enforce access control at world granularity, which is not sufficient to isolate mutually distrusting peripherals. The RMM isolates mutually distrusting Realm VMs using stage-2 translations, which are stored in the protected Realm memory (EL2) inaccessible to the host OS and the VMM. Similarly, to enable the secure device I/O at device granularity, PORTAL uses the SMMU stage-2 translations to isolate mutually distrusting devices from accessing each other's PORTAL regions. Since the isolation guarantee solely depends on correctly identifying the world to which the VM and the device belong, the identity of VMs and devices must be unforgeable. PORTAL uses the VMID of each VM and the SID of each device to identify VMs and devices, respectively. Both identifications belong to the SMMU data structures and are protected by the dedicated PORTAL region as described in §5.2.2. Moreover, the configuration of the SMMU is managed by the PORTAL System Realm instead of the untrusted host OS and the VMM.

**Isolating authorized and unauthorized entities.** PORTAL designs two types of custom GPTs, PORTAL-GPT (P-GPT) and Normal-GPT (N-GPT), to configure the memory views for authorized and unauthorized entities, respectively, as shown in Figure 5.2. The P-GPT of an authorized entity (*i.e.*, a Realm VM or a device) sets its PORTAL region as the Normal world PAS. For a Realm VM on a CPU core, its P-GPT maintains the PAS layout of memory regions, excluding the PORTAL region as in its original GPT, preventing untrusted software from accessing the Realm and Root regions. Isolation among distrusting Realm VMs is enforced by the RMM's stage-2 translation. For a device attached to the Realm VM, the P-GPT in the SMMU configures non-PORTAL regions as Root world PAS, restricting its access to only the PORTAL region for DMA. Isolation among distrusting devices accessing any PORTAL region is enforced by SMMU stage-2 translation, ensuring each device accesses only its own PORTAL region. In contrast, the N-GPT of an unauthorized entity (*i.e.*, a Realm VM or a device) configures the PORTAL regions as Root world PAS to block unauthorized access while preserving the original access control of other memory regions. Consequently, memory transactions from unauthorized entities to the PORTAL regions are bound by lower exception levels and rejected by GPC before reaching the memory controller.

*Switching between P-GPT and N-GPT in CPU.* Each CPU core's GPT is set as a P-GPT for the authorized Realm VM running on it. When not executing a Realm VM, the P-GPT transitions to an N-GPT to block unauthorized access to the PORTAL region. The Monitor in the Root world intercepts Realm Management Interface (RMI) calls (*e.g.*, `RMI_REC_ENTER`) that start a Realm VM, transforming the N-GPT to a P-GPT before switching from the Normal to the Realm world, and vice versa (Figure 5.2).

*Synchronizing P-GPT and N-GPT.* Regardless of the access control on PORTAL regions, N-GPT and P-GPT must provide the same access control between different PAS as per CCA policies. Except for PORTAL regions, P-GPT and N-GPT should be identical for other memory sectors. Updates requested via RMI and Realm Service Interface (RSI) interfaces on the GPT should be applied to both P-GPT and N-GPT for any Realm page not in a PORTAL region.

**Isolating MMIO in Realm VMs.** Given that MMIO controls integrated devices (*e.g.*, GPU and SMMU) in SoC, granting exclusive MMIO access to authorized Realm VMs

**Table 5.1:** Updated and new interfaces introduced by PORTAL. `RMI_REC_ENTER` is an existing interface and is updated for PORTAL-specific operations.

| API | Description |
|---|---|
| RMI_REC_ENTER* | initiate the execution of a Realm VM.<br>PORTAL updates P-GPT and N-GPT.<br>PORTAL check the device attachment / detachment. |
| RSI_ATTACH_DEV | allocate a device from a Realm VM. |
| RSI_DEV_MNG | attach and detach a device from Realm world. |
| RSI_SET_PORTAL | request a PORTAL region for DMA. |
| RMI_ATTACH_DEV | allocate device memory from Normal world to Realm world. |
| RMI_CREATE_Q | create a command queue for a newly instantiated Realm VM. |
| SMC_SET_PORTAL | delegate Realm memory to a PORTAL region.<br>add stage-2 translation for the SMMU. |

effectively protects each device. Since MMIO uses physical memory addresses to map peripheral registers, PORTAL employs GPC protection for access control on MMIO pages and host them in the Realm VM's PORTAL region. However, Arm CCA does not validate the mapping between the guest Physical Address (PA) and the host PA by default when adding new memory pages to the Realm VM. Each platform has unique and fixed MMIO physical addresses for devices, listed in the device tree and immutable at runtime. PORTAL can verify that the MMIO page mappings match these fixed addresses. Without verification, attackers can map the guest PA to a malicious host PA, enabling man-in-the-middle attacks.

### 5.2.4 System Realm Internals

We adhere to the Arm CCA specification for RMM communication with the untrusted host OS and VMM [37] and propose new interfaces (Table 5.1). The interfaces `RSI_ATTACH_DEV` and `RMI_ATTACH_DEV` allocate devices for Realm VMs, while `SMC_SET_PORTAL` configures the device under PORTAL protection.

**System Realm initialization.** We illustrate the initialization of the System Realm for exclusive SMMU management in Figure 5.3. Assigning a device (*i.e.*, SMMU and peripheral) to a Realm VM is achieved by providing it exclusive access to the MMIO region of the device. Since the host VMM manages all memory resources, including SMMU MMIO addresses, the System Realm invokes `RSI_ATTACH_DEV` on the SMMU address to transfer the SMMU device from the host VMM to the System Realm (❶). As PORTAL allows only

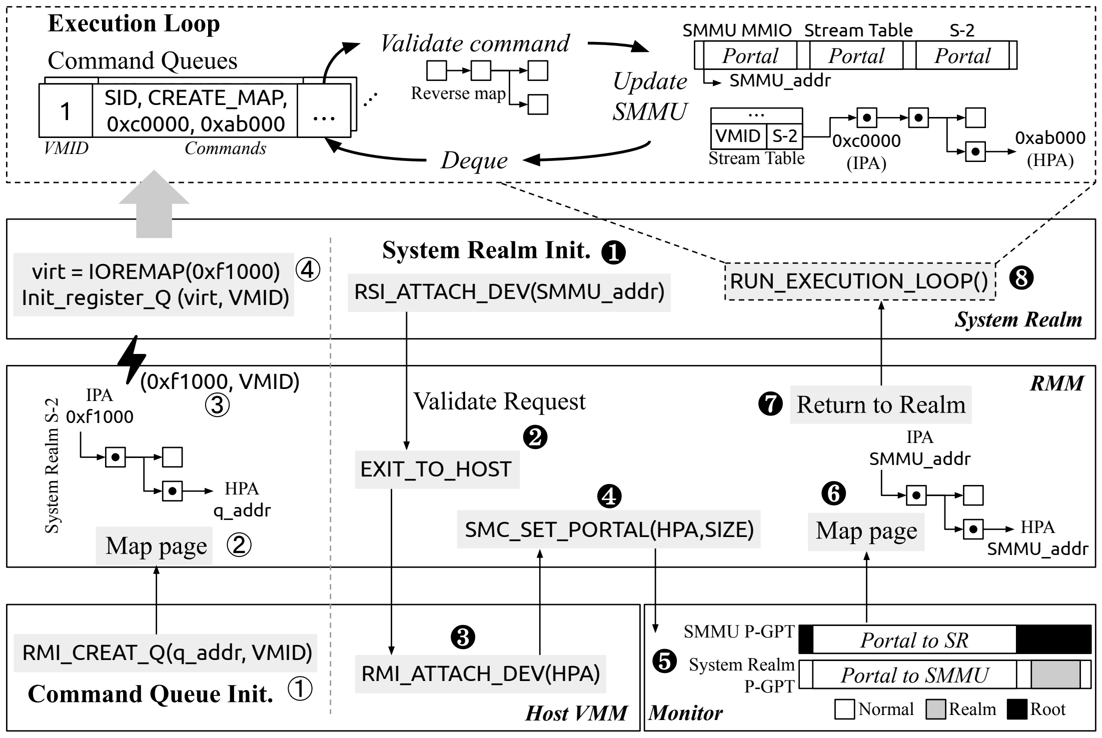

**Figure 5.3:** System Realm lifecycle.

the System Realm to acquire the SMMU, the RMM validates the `RSI_ATTACH_DEV` call by checking the requesting Realm VM's measurement against the System Realm (❷). If the System Realm initiates the SMMU device attachment request, the host delegates the PA pages mapped to the SMMU to the RMM. That is, the host detaches the SMMU driver and delegates the memory pages mapped to the SMMU MMIO region to the System Realm via `RMI_ATTACH_DEV` (❸). Since the host is not trusted, the delegated host PA pages are checked against the requested addresses. If they match, the RMM invokes `SMC_SET_PORTAL` (❹) to configure the pages within a PORTAL region for the System Realm (❺) through the Monitor, which updates the P-GPT and N-GPT accordingly. After the RMM creates a mapping in the stage-2 page table of the System Realm (❻), it returns to the System Realm (❼), which then initiates an execution loop (❽) to exclusively process the commands submitted by Realm VMs to manage the SMMU.

**Command queue for efficient command submission.** To reduce inter-realm context switching overhead for SMMU management, PORTAL uses ring buffers between the System Realm and other Realm VMs as command queues for efficient command submission, as shown in Figure 5.3. When a Realm VM is instantiated, users can set a flag to indicate PORTAL support. Based on this flag, the host VMM invokes `RMI_CREATE_Q` to create a command queue for the Realm VM. The host then delegates the command queue memory pages to the RMM (①). The RMM maps the command queue in the stage-2 page table, ensuring exclusive access for the RMM and System Realm (②). An exception is raised to the System Realm (③) to notify it of the new command queue. A pre-registered exception handler establishes the virtual-to-physical address mapping for the command queue, allowing access by the System Realm during execution. After registering the command queue, the System Realm returns from the exception and continues the execution loop (④).

**Execution loop for command handling.** After initializing the System Realm and command queues, the System Realm dequeues pending requests to process SMMU configuration requests for the requesting Realm VMs (Figure 5.3). Under benign scenarios, to establish a PORTAL region between a Realm VM and a device, the Realm VM requests the System Realm to map its PORTAL region into the SMMU page table. To prevent unauthorized mappings, the System Realm verifies 1) the host PA of the PORTAL region and 2) the owner of the PORTAL region (*e.g.*, VMID of the Realm VM). Each command queue is a dedicated channel between the System Realm and a Realm VM, making it easy to identify the source of requests (*i.e.*, the VMID of the requesting Realm VM). Requests are submitted by a Realm VM through an RMI call into the RMM, and the RMM holds the VMID information of the requesting Realm VM. By comparing the VMID of the requesting Realm VM with the VMID that owns the target PORTAL region, PORTAL verifies if the requesting Realm VM has the appropriate permissions to configure a specific PORTAL region for the peripheral device in the SMMU (⑤). To aid in verification, the System Realm maintains a *reverse mapping* to verify that the requested physical addresses have not been already mapped in the other I/O page table to establish the PORTAL region. After verification passes, the System

Realm updates the SMMU metadata, such as the Stream Table, to generate mapping to allow authorized devices to access the PORTAL region (⑥).

**On-demand command handling.** Keeping the System Realm running constantly wastes CPU resources when idle. Instead, it is activated on demand for SMMU-related tasks. However, frequent context switching between Realm VMs and the System Realm can still occur. For example, devices like GPUs need memory pages for metadata, such as command and interrupt queues, which are only used after device initialization. Thus, invoking the System Realm for each allocation request of these pages is inefficient. In addition, since the host VMM [218] manages inter-realm context switching, additional overhead between the Realm and Normal world is inevitable. To reduce overhead, PORTAL *lazy-loads* the System Realm when a device encounters a page fault from invalid DMA mappings in the SMMU. PORTAL defers SMMU management until the device accesses the PORTAL region for DMA, avoiding excessive context switching during initialization.

### 5.2.5 Direct Memory Access (DMA)

Devices and the host use DMA to transfer large amounts of data. After a device is attached to a Realm VM, memory should be allocated for DMA and protected by PORTAL for secure transmission. Unlike MMIO, which has a fixed address in the host PA and is vendor-configured, the DMA region is dynamically generated and configured at runtime. Thus, a PORTAL region for DMA cannot be pre-allocated in the Realm VM like MMIO but must be requested and configured during runtime.

To allocate a PORTAL region for DMA, the requesting Realm VM submits a request to the System Realm by calling `RSI_SET_PORTAL` to change the granule of the desired physical pages from Realm PAS to Normal world PAS. The Realm VM sends a list of guest PA with size along with the device id to the RMM. The RMM fulfills this request by translating the guest PA to host PA using its stage-2 translation tables for the Realm VM. Note that the Realm memory transitioned to the Normal world PAS as the PORTAL region already belongs exclusively to the requesting Realm VM, ensuring intra-realm isolation. After the granule changes, only the requesting Realm VM can access the PORTAL region as plaintext memory, while other Realm VMs and untrusted peripherals are prohibited by the N-GPT. Besides CPU configuration, the SMMU page table must be updated for peripherals to access the PORTAL region via DMA. To transition device memory to the initialized PORTAL region, the Realm VM requests the System Realm to update the SMMU 's stage-2 tables (§5.2.3). With P-GPT configured, the device can access the PORTAL region via DMA after proper SMMU stage-2 mappings. This approach only requires kernel modification for DMA setup, without changes to the application, device driver, runtimes, or device logic.

### 5.2.6 Device Management

Unlike the traditional VM model in which a device is dedicated to a Realm VM throughout its lifetime, under the SoC model, multiple Realm VMs need to share the device. Never-However,current CCA design, utilized by existing studies [79, 80], incorporates the device in the initial attestation of the Realm VM, which forces the device to be bond to the Realm VM for its entire lifespan. To support runtime device management, PORTAL introduces per-device states managed by the RMM.

Supporting runtime device management securely is non-trivial. Before deeming the ownership transfer of a device from one Realm VM to another is completed and the device is ready for use, not only the stage-2 page table mapping on the CPU side, but also the I/O page table of the SMMU should be completely transferred to the destination Realm VM. For example, assume that Realm$_B$ wants to attach a device occupied by Realm$_A$. If the device is deemed ready for use after updating the SMMU I/O page table, but the stage-2 page table mapping on the CPU side remains unchanged, then Realm$_A$ can still interact with the device, which will now have access to memory pages of Realm$_B$. This breaks the isolation between Realm VMs and is detrimental when Realm$_A$ is malicious.

We illustrate how PORTAL achieves runtime device management securely in Figure 5.4.

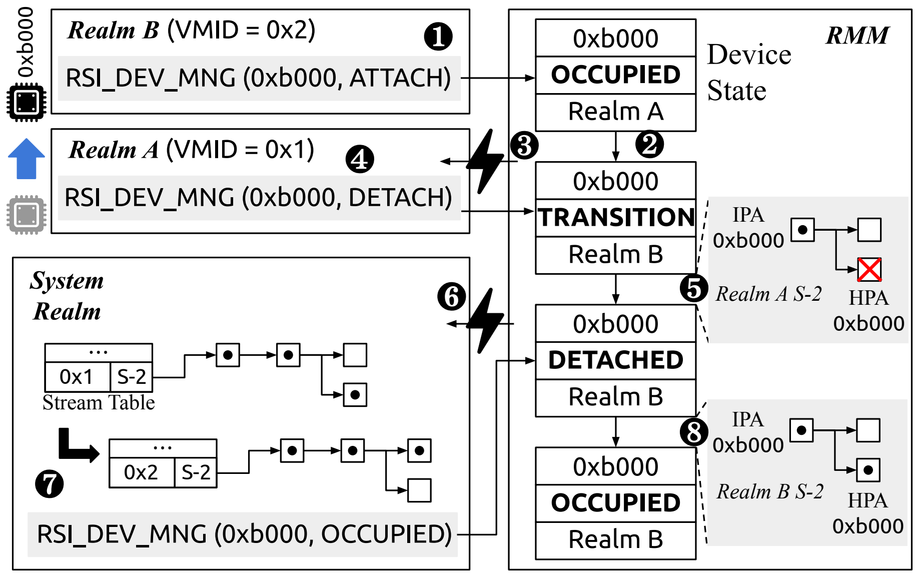

**Figure 5.4:** Device management of PORTAL.

The device attachment request is initiated from Realm$_B$ through `RSI_DEV_MNG` (❶). It requires the base address of the device's MMIO region and a command to notify RMM that it wants to attach a new device. Since the device has been owned by Realm$_A$, RMM first updates the state of the device from `OCCUPIED` to `TRANSITION`. The RMM also updates the owner to indicate that the device will be transferred to Realm$_B$ (❷). The RMM then enters Realm$_A$ by injecting an interrupt so that Realm$_A$ can handle device detachment operation as it is the current owner of the device (❸). Realm$_A$ handles device detachment and invokes `RSI_DEV_MNG` to notify the RMM that it is safe to detach the device from Realm$_A$ (❹). The RMM destroys the mapping to the device MMIO pages in the stage-2 page table of Realm$_A$ to prevent its further access to the device. Meanwhile, the RMM updates the device state to `DETACHED` (❺). In order to request the System Realm to update device ownership in the Stream Table, the RMM injects an interrupt to the System Realm (❻). The System Realm switches the I/O page table pointer stored in the Stream Table and updates VMID to initiate device support for Realm$_B$. Also, it invokes `RSI_DEV_MNG` to notify the RMM of the completion of the SMMU update (❼). Finally, the RMM updates the device state to `OCCUPIED` and updates the stage-2 page table of Realm$_B$ so that the Realm$_B$ accesses the device exclusively (❽).

**Device Attestation.** If the device remains attached until the Realm VM is destroyed, it can be attested during Realm VM initialization. However, the initial measurement value will not account for runtime device attachment and detachment events. Therefore, PORTAL makes use of the Realm Extensible Measurement (REM) register in Arm CCA to track device management histories maintained by the RMM. When the status of the device managed by PORTAL changes, PORTAL logs the changes and updates the REM register to measure the logs. The device status can only be updated through the RMI/RSI calls, so the attackers cannot manipulate the log directly. Furthermore, even though attackers can invoke RMI/RSI calls to detach devices from the Realm VM or prevent their assignment, the entire device management log is securely maintained with its measured hash. This allows the victim to analyze and triage the attacks afterward. The only feasible attack against device management is a denial-of-service attack, which is considered out of scope.

## 5.3 Implementation

We implement PORTAL in two types of prototypes: 1) a functionality prototype on Arm FVPs that verifies the design and security of PORTAL, and 2) a performance prototype that uses Armv8 instructions to emulate the latency of Arm CCA features in the coming Armv9.

### 5.3.1 Functionality Prototype

PORTAL is prototyped on Arm FVP_Base_RevC-2XAEMvA [112] with RME support. The FVPs simulates the Arm system, including processors, memory, and peripherals. Nevertheless, the FVPs does not support genuine unified-memory devices found on commercial SoC Arm chips. Instead, the FVPs includes a connected test engine that handles peripheral memory accesses as if it were a DMA-capable peripheral, along with an SMMU that supports RME. We use the simulated devices to verify the isolation guarantee of PORTAL. We use Linux v5.1 kernel as the host OS and the TF-A v2.10 as the Monitor. To verify the isolation guarantees of PORTAL, we initialize GPTs in the FVPs prototype and test with the emulated CPU and the test engine peripheral. As the existing TF-A assigns the same GPT for both components, we additionally reserve a 0.5MB memory for device GPTs. To use the GPTs to control the data transmission between the CPU and the peripheral, we configure the system registers of MMU and SMMU. We also provide a reference implementation of the System Realm, which in reality should be mediated and distributed by trusted parties such as Arm and processor vendors.

### 5.3.2 Performance Prototype

Given that the FVPs does not provide cycle-accurate emulation [112, 78], we developed a physical prototype based on Armv with Armv9 CCA features to evaluate PORTAL's performance. We migrated our FVPs prototype to the Orange Pi 5 Plus [113], equipped with an RK3588 SoC [219] featuring an 8-core 64-bit Arm processor (4-core A76 and 4-core A55). The SoC includes an Arm Mali-G610 GPU and 8GB of DRAM shared between the processor and the GPU. We run a customized TF-A to emulate Arm CCA. We run Ubuntu 24.04 with Linux v6.1 kernel as the Normal world host. User Realm VMs and the System Realm run a customized Linux v6.2 kernel. We disabled the A55 cores and used the A76 cores to measure overhead reliably.

**Emulation of Arm CCA.** We replace all CCA instructions and registers with Armv8 features to simulate CCA. First, TF-A (*i.e.*, the trusted Monitor) retrieves the GPT hardware configuration from `GPCCR_EL3`. We replace it with instructions returning a fixed value to ensure GPT initialization without exceptions. In addition, the ARMv8 processor differentiates only between Secure and Normal worlds, lacking Realm world support. We created a Realm context in the Normal world and patched the TF-A to validate the security state origin (*e.g.*, Normal or Realm) of SMC calls for proper RSI handling from the Realm VM. Lastly, we moved the GPT initialization code (*i.e.*, TF-A `bl2`) to `bl3` since the boot sequence of the evaluation board (*i.e.*, the Orange Pi 5 Plus) is designed to run `u-boot` secondary program loader as its second-stage bootloader instead of TF-A `bl2`.

## 5.4 Evaluation

In this section, we evaluate our PORTAL prototypes according to the following research questions:

- **RQ1:** How large is the TCB of PORTAL?
- **RQ2:** Can PORTAL defend against privileged adversaries?
- **RQ3:** How much performance benefit does PORTAL provide?
- **RQ4:** How much performance overhead does PORTAL incur?
- **RQ5:** How much memory overhead does PORTAL incur?

We measure the performance benefit of our prototype using the Rodinia benchmark suites [220], covering various Arm GPU use cases. We also evaluate the performance overhead of PORTAL due to its lifecycle, the System Realm, and device management. Each experiment is conducted 10 times, and we calculate the average values as the final results.

### 5.4.1 RQ1: TCB Size

**Table 5.2:** Introduced TCB size of PORTAL measured in LoC.

| Layer | Lines of Code (LoC) |
|---|---:|
| TF-A | 96 |
| RMM | 784 |
| System Realm | 550 |
| Realm VM (Guest kernel) | 230 |
| All | 1,660 |

We measure the TCB size of PORTAL using cloc [221] in terms of standard lines of source code. PORTAL introduces 1,660 LoC additions in total.

### 5.4.2 RQ2: Security Analysis

**Unauthorized memory access and modification.** To compromise data security, an attacker may directly leak or manipulate the data transmitted between the Realm VM and the peripheral. PORTAL utilizes Arm CCA's GPC to protect the shared PORTAL region from unauthorized access by entities such as the host OS, VMM, and untrusted peripherals. In particular, the PORTAL region is visible as Normal world PAS only to authorized Realm VMs and peripherals, whereas it is seen as Root world PAS by all other entities. This configuration is set in GPTs and is enforced by GPC. An attacker might also directly request a Realm VM under her control to directly access the sensitive data in other Realm VMs. However, such a request fails to bypass the memory isolation on the CPU side due to stage-2 translation in the RMM. Same attacks using malicious applications on the device side fails for the same reason due to the SMMU GPC.

**Illegal device management.** Given that device memory is managed by the untrusted hypervisor, a malicious hypervisor could attempt to expose sensitive information by dishonestly handling memory requests (*e.g.*, Iago-style [222] attack). To defend against these attacks, the System Realm ensures that the device memory does not overlap with the memory allocated to other realms and devices. Specifically, the System Realm verifies whether the PORTAL region used for the device communication overlaps with the Normal world PAS of other entities in the GPTs. The attacker might also attempt to deliberately map the device memory to an unauthorized area or create duplicate mappings. In such scenarios, the System Realm checks the mappings before making changes to the actual device page table (*i.e.*, the SMMU stage-2 page table).

**Fake device and SMMU.** An attacker (*e.g.*, VMM) could emulate a fake peripheral device and attempt to transfer sensitive data into this simulated device. In addition, the attacker could also simulate the SMMU to compromise the GPT isolation guarentees. PORTAL guarantees that the CCA Monitor communicates with actual hardware devices rather than simulated ones. PORTAL takes advantage of the fact that the physical addresses of Arm peripheral devices and SMMU MMIO registers are fixed and unmodifiable in most Arm-based devices [223, 224, 225, 226]. Consequently, malicious or emulated devices must be assigned to different fixed addresses, enabling PORTAL to confirm that the data transfer is occurring with the genuine device.

**Malicious co-tenants.** An attacker can deploy a malicious Realm VM that is co-located with the victim Realm VM on the same platform. The malicious Realm VM may attempt to allocate DMA regions that overlap with the victim's devices to gain access to sensitive data. In PORTAL, the System Realm ensures that the malicious Realm VM cannot request DMA mappings on PA that it does not own by verifying the stage-2 translation table, thereby preventing overlapped DMA regions.

**CPU GPC bypass.** An attacker might attempt to circumvent the CPU GPC to gain access to restricted memory. One possible method is to disable the GPC or replace the GPT with a malicious version. However, since the GPC registers are located in the Root world, the attacker lacks the necessary privileges to access them. The attacker might also try to directly alter the GPT to compromise memory isolation, but this is also thwarted as the GPTs resides in the Root world. Finally, the attacker might exploit GPC TLB entries to access the protected region. That is, because the TLB is cached following the GPC, when the GPT is updated, the TLB will continue to correspond to the GPC based on the old GPT version. PORTAL mitigates such attacks by ensuring that TLB entries are flushed whenever the CPU GPT is modified.

**Device GPC bypass.** Just as with CPU GPC, attackers may attempt to circumvent the GPC on the device side to gain unauthorized access to the device memory, other realms, and the TCB of PORTAL. Nevertheless, the attacker is unable to directly access the registers that control GPC because she does not possess Root world privileges. Furthermore, PORTAL protects the device GPTs within the Root world, preventing the attacker from directly altering them to undermine access control. Similarly, the attacker could exploit the TLB to circumvent GPC. Thus, PORTAL ensures that TLB entries in the SMMU are flushed whenever there are changes to the device GPT.

**Malicious DMA.** An attacker could exploit other devices to gain unauthorized access to the PORTAL region for DMA operations between the Realm VM and the connected

**Table 5.3:** Configuration of the selected Rodinia benchmark

| Application | Problem Size | Data Buffers | Memory |
|---|---|---|---|
| `Gaussian` | 1024 × 1024 nodes | 3 | 8.39 MB |
| `LUD` | 2048 × 2048 nodes | 1 | 16.00 MB |
| `Pathfinder` | 100000 × 100 points | 4 | 40.46 MB |
| `NW` | 2048 × 10 nodes | 2 | 16.79 MB |
| `Hotspot3D` | 512 × 512 × 8 nodes | 3 | 25.16 MB |
| `NN` | 42764 nodes | 2 | 0.51MB |

device. Although a PORTAL region is a plaintext shared memory area, PORTAL depends on the GPT's stage-2 translation table to block untrusted devices from accessing the PORTAL region designated for DMA. Specifically, the PORTAL region currently in use is configured as Root world PAS in the GPTs for untrusted entities. As the attacker might instead launch a malicious application on the victim's device to achieve the same goal, PORTAL ensures that when a device is assigned to other realms, a PORTAL region that is not overlapped with other PORTAL regions is assigned for DMA.

**Physical Attacker.** A physical attacker can take advantage of a compromised device (*e.g.*, with debugging features enabled or loaded with vulnerable firmware). When such a device is connected to a Realm VM, it can compromise the platform's security. PORTAL depends on the remote verifier to confirm that the device connected to the Realm VM is correctly configured during Arm CCA remote attestation. Moreover, a physical attacker might attempt to directly probe the memory related to PORTAL regions to extract sensitive information. However, this type of attack is not viable under PORTAL's threat model, which relies on the robustness of SoC-based Arm processor packages. Additionally, we assume that probing or maliciously redirecting the memory transactions of PCIe peripherals is prevented by the security features of Integrity and Data encryption (IDE) included in PCIe-5 [227].

### 5.4.3 RQ3: Performance Benefit of PORTAL

We evaluated the performance of GPU tasks from the Rodinia benchmark both under the traditional confidential model with memory encryption and PORTAL with plaintext isolated memory for DMA. We adopt AES-GCM encryption on both CPU and GPU to establish

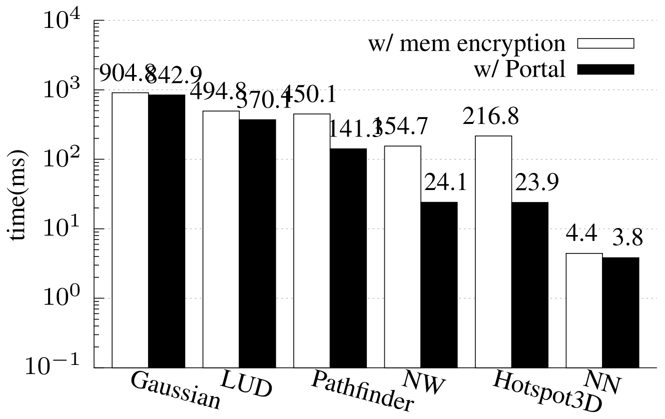

**Figure 5.5:** Performance of GPU tasks when protected by memory encryption and by PORTAL.

a baseline where Realm VM and the device communicate via encrypted shared memory for DMA. We report the problem size and memory consumption in Table 5.3. As hown in Figure 5.5, the performance overhead due to memory encryption is heavier when the processing logic is simpler but the data size is larger, which conform to the characteristic of memory encryption. It also shows that PORTAL benefit the most when protecting data intensive applications where larger size of data is transmitted frequently. Overall, PORTAL achieves an average performance improvement of 3.71× (1.07×-9.07×) over the six selected Rodinia benchmark GPU tasks thanks to its plaintext isolated memory for device communication. Note that, the hardware-based encryption adopted in the upcoming RME should have a much better performance compared to AES-GCM. However, eliminating the memory encryption of RME should still grant a noticable performance benefit for PORTAL.

### 5.4.4 RQ4-1: Performance of PORTAL Lifecycle

**Initializing GPTs.** PORTAL uses two GPTs to protect regions (*i.e.*, P-GPT and N-GPT), initialized at boot time. The existing system-wide GPT is used as N-GPT, and P-GPT is additionally populated. Initializing the additional GPTs incurs a one-time overhead.

**Initializing Realm VM.** PORTAL incurs minimal overhead during Realm VM creation if the PORTAL flag is not set. When PORTAL is activated, additional overhead occurs due to invoking `RMI_CREATE_Q` to establish the command queue page. Note that `RMI_CREATE_Q` can be invoked at any point during Realm VM creation, as injecting exceptions and handling the command queue can occur concurrently. Realm VM creation time varies with its size, but no noticeable overhead is detected if `RMI_CREATE_Q` is invoked early. Invoking `RMI_CREATE_Q` at the end of creation introduces a 2% overhead.

**Entering Realm VM.** PORTAL adds a conditional check in `RMI_REC_ENTER` to address device management before entering the Realm VM. The runtime overhead, regardless of PORTAL usage, is a single-bit flag check in the Realm Descriptor, equivalent to 5 instructions. If there are pending device management operations for Realm VM, the RMM injects a fault to notify Realm VM. The injection process involves reading system registers, determining the handler address, and updating the system register to resume execution from the exception handler. This injection overhead is 6.3% compared to executing `RMI_REC_ENTER` with the device management flag unset.

### 5.4.5 RQ4-2: Performance Overhead of System Realm

**Initializing System Realm.** The System Realm is instantiated during host OS boot and persists until system shutdown. PORTAL incurs a one-time 1.5% overhead for the System Realm during boot, which involves Realm VM creation, SMMU device attachment, and data structure setup (*e.g.*, Stream Tables). Most of the overhead arises from Realm VM creation, not from the additional RMI/RSI calls for SMMU attachment.

**Processing SMMU management commands.** PORTAL incurs two types of overhead in processing commands. The first is the cost of engaging the system realm when an SMMU interrupt occurs due to an invalid mapping in the PORTAL region. This involves handling the SMMU interrupt in the VMM and forwarding the virtual interrupt to the System Realm. Since CCA supports interrupt injection through existing `RMI_REC_ENTER`, PORTAL does not require additional operations to engage the System Realm. It takes 2 µs, with most of the overhead from world switching incurred by entering the System Realm. This can be minimized by allocating a dedicated physical core to handle commands in a polling style. The other overhead involves processing commands, which includes traversing queues, checking reverse mappings, and creating SMMU I/O page table mappings. In our evaluation, two Realm VM instances using PORTAL generated 10 commands per queue, requesting to insert SMMU entries. Processing 20 commands took 1.9 µs to 2.4 µs. We dedicated one core for the execution loop in the System Realm. This overhead can be optimized by assigning multiple cores as the number of command queues increases.

### 5.4.6 RQ4-3: Performance Overhead of Device Management

**Device initial attachment.** The overhead of device attachment varies based on the device's status requested by the Realm VM. If the device is not assigned to a Realm VM, its MMIO pages must be first delegated to the RMM and configured as a PORTAL region via `SMC_SET_PORTAL`. Our evaluation shows that attaching the MMIO pages of the Mali-G610 GPU (512 pages) takes 46.29 µs.

**Device reassignment.** If the device is occupied by another Realm VM, delegating MMIO pages is unnecessary. Instead, there are extra communication costs between the current Realm VM and the System Realm, as detailed in Figure 5.4, incurring 175 µs overhead. Note that most of the overhead comes from multiple world switches due to inter-Realm VMs communication. While device attachment is costly, it is not frequent. In addition, in contrast to the current RMM design [218], which involves the VMM for each MMIO page access, PORTAL assigns MMIO pages directly to the Realm VM's PORTAL region, eliminating additional host involvement.

### 5.4.7 RQ5: Memory Overhead

**Maintaining P-GPT.** In our evaluation, the protected memory region is configured as 4GB. The GPT uses two levels of page tables: 1GB for the first-level and 4KB for the last-level. Adding one GPT as P-GPT requires a 4KB page in SRAM (first-level GPT) and 0.5MB of DRAM (second-level GPT). Compared to existing approaches [80], PORTAL does not need additional GPT per device because the SMMU provides extra protection using the P-GPT configured for SMMU, enabling PORTAL to be used in constrained devices with limited memory.

**Device management & Command queue.** The RMM must maintain per-device data structures for secure device management. Considering the device status and the owner, the next transition state and the authorized Realm VM to initiate the transition can be determined. Since PORTAL does not trust the VMM for device management, per-device status is stored within the RMM. PORTAL allocates 8 bytes for each device: 40 bits for Realm Descriptor address (page-aligned), 16 bits for VMID, and 2 bits for device state. PORTAL also allocates a 256KB command queue per Realm VM, including VMID, locks, and the queue. With a 16-byte entry size, up to 16,384 requests can be queued per Realm VM. The command queue size is based on the experiment in §5.4.5.

## 5.5 Related Work

**Secure Device Access in Arm CCA.** Strongbox [77] enhances TrustZone-based isolation for integrated GPUs on Arm platforms, excluding the driver from the TCB. However, StrongBox trusts the Secure World, which PORTAL excludes from its TCB to align with Arm CCA's threat model. StrongBox is also incompatible with hypervisors, which, if compromised, can bypass the stage-2 translation, PORTAL's primary protection mechanism. Compared to StrongBox, PORTAL is more suitable for next-generation Arm devices. Recently, Arm CCA has introduced a proposal to support peripheral devices. However, this support, known as RME Device Assignment (RME-DA) [37], remains a theoretical concept without finalized hardware implementation. To address the issue of data security in external peripherals (*e.g.*, PCIe), a recent study called ACAI [79] suggests a similar design to RME-DA by extending CCA security invariants to device-side access. Compared to these designs, PORTAL has a more compact TCB. Both RME-DA and ACAI add additional management logic to the privileged RMM (EL2) and Monitor (EL3). PORTAL utilizes a dedicated System Realm for device management tasks, preventing the TCB of CCA core components from becoming bloated. In addition, RME-DA introduces hardware changes on the SMMU and ACAI replaces the SMMU configuration code with special interfaces. Instead, PORTAL does not introduce any modification to the SMMU by relying on the Normal World PAS for isolated memory, and the System Realm for exclusive SMMU management. CAGE [80] seeks to enhance unified-memory GPU support on Arm CCA. CAGE introduces a novel shadow task mechanism to flexibly manage confidential GPU applications and utilizes multiple GPCs to achieve two-way isolation. CAGE aims to efficiently execute confidential GPU tasks on Arm CCA, while PORTAL is designed to provide fast and secure I/O for a wide range of Arm peripheral devices.

**GPU TEEs.** GPU TEEs have been proposed recently to secure the adata security o GPUs. In order to establish a secure I/O between the user and the GPU, a typical GPU TEE relies on the CPU-side TEE to transfer the sensitive data to the GPU and manage the access control to the GPU MMIO. For example, HIX [228] and HoneyComb [229] use Intel SGX and AMD SEV to secure the GPU MMIO, respectively. Cure [61] secures the GPU TEE by adding an access control filer in the system bus. HETEE [230] protects the GPU resources within isolation environments by introducing a security controller on the FPGA hardware. Due to the difference in architecture and chip design, these approaches might not be suitable to adopt these approaches to protect Arm GPUs. Recent works have also proposed Arm-based TEEs for GPUs [77, 231, 232, 233]. These approaches utilize traditional Arm security primitives, such as the Arm TrustZone, while they provide in sufficient security guarantees under the security model of Arm CCA. Lastly, several efforts have been made to directly house TEEs in side GPUs without the CPU-side TEEs, including Graviton [234] and Nvidia H100 GPUs [44]. The standalone GPU TEEs monitor the commands and data submitted by the host and ensures confidentiality during task execution. However, these approaches rely on the fact that the GPU memory and the host are naturally isolated as extarnal GPUs possess private memory, while such assumptions do not hold for integrated GPUs on Arm processors with unified memory model.

## 5.6 Discussion

**Limitations.** PORTAL relies on the assumption that the SoC package is physically secure, making physical memory attacks infeasible. While this is valid for mobile SoCs, one of the largest Arm markets, extending PORTAL to other platforms where this assumption does not hold presents a challenge. Potential solutions could involve integrating additional physical security measures or incorporating selective memory encryption where necessary. PORTAL also requires vendors to implement and distribute a System Realm tailored to their specific SoC configurations. This increases the burden on vendors, who must ensure the System Realm's correctness and security through meticulous validation and testing. Collaborating with Arm and industry stakeholders to develop standardized implementations of the System Realm could help mitigate this burden.

**Discrete devices.** PORTAL's utility extends beyond SoC environments to include scenarios where data must be transmitted from SoC integrated memory to external peripherals connected through PCIe, such as discrete GPUs. In these cases, PORTAL can leverages the link-layer encryption provided by PCIe-5 [227], which ensures secure data transmission between the SoC and the external devices. This approach requires only a one-time encryption provided by the PCIe link layer, eliminating the need for additional software-based or hardware-based encryption on the SoC. This not only maintains the performance and power efficiency benefits of PORTAL but also enhances its applicability to a broader range of devices and configurations. By relying on PCIe link-layer encryption, PORTAL can effectively secure data in transit without the overhead associated with traditional encryption methods, making it a versatile solution for both integrated and discrete peripheral environments.

**Regulatory and compliance considerations.** The adoption of PORTAL in real-world applications must navigate various regulatory and compliance considerations, particularly in industries with stringent data protection requirements. While PORTAL enhances performance and power efficiency by eliminating memory encryption, some applications may still require encryption to comply with regulations such as GDPR, HIPAA, and PCI-DSS. To address this, PORTAL could integrate optional encryption mechanisms to meet these specific regulatory standards without compromising its core benefits. Additionally, the implementation of the System Realm by vendors necessitates thorough validation to ensure compliance with security certifications and standards. Engaging with regulatory bodies and industry consortia to establish PORTAL as a compliant solution across different sectors will be crucial.

# CHAPTER 6
# CONCLUSION

The evolution of cloud computing, IoT, edge computing, and emerging platforms has shifted data control, requiring users to entrust their digital information to third-party providers, raising significant data privacy and security concerns. Trusted Execution Environments (TEEs) offer a solution by establishing secure enclaves in processors to protect sensitive data. While TEEs have been successfully integrated into IoT and edge computing, their adaptation to rapidly developing platforms is still pending. This thesis focuses on integrating TEEs with emerging mobile platforms, addressing the unique security challenges and high performance demands of mobile platforms.

Current TEEs are not immune to security threats, notably side-channel attacks (SCAs) that exploit shared resources like cache memory to extract secret data. Modern TEEs like Intel SGX, AMD SEV, and Arm TrustZone aim to protect against privileged attackers but lack practical and fundamental solutions to SCAs.

One the one hand, multiple SCAs can co-exist on a target platform, while defending against multiple SCAs simultaneously is a non-trivial task. This thesis introduces PRIDWEN, a novel framework that dynamically synthesizes a secure SGX program that is optimally hardened against various SCAs simultaneously, while preventing any deployability, redundancy, or incompatibility problem. To overcome the restrictions of the static deployment model of SGX, PRIDWEN adopts Wasm as the IR, and supports smooth integrations of instrumentation passes for both hardware-assisted and software-only mitigations. PRIDWEN selects an optimal set of mitigations to be applied at runtime according to the hardware configurations of the target platform, and provides means for the user to validate and attest the final synthesized binary. We implement a prototype of PRIDWEN, which integrates four SCA defenses. Through extensive evaluation, we show that PRIDWEN efficiently hardens

SGX programs with chosen defenses, while incurring moderate performance overhead.

On the other hand, current SCA detection techniques within TEEs exhibit various limitations due to the absence of direct access to precise microarchitectural information and the lack of flexible methods to respond to potential SCAs inside TEEs. This thesis introduces SENSE, an innovative interface that empowers TEE programs to directly subscribe to microarchitectural event notifications and actively manage these events using userspace handlers. We present a comprehensive design of SENSE for cache and conduct an in-depth security analysis to demonstrate its robustness. The evaluation reveals that SENSE effectively mitigates side-channel attacks while maintaining minimal performance overhead.

Finally, maintaining performance and energy efficiency requirements is crucial for emerging mobile platforms. Any adaptation of TEEs must enhance security without compromising such requirements. Mobile devices predominantly use Arm's CPU architecture, which has evolved to integrate diverse components like GPUs and NPUs, supporting the demands of modern mobile applications. However, Arm CCA, a recent advancement in confidential computing, faces challenges in keeping up with the integration trend in mobile Arm SoCs. PORTAL provides a novel solution for secure and efficient device I/O in Arm CCA on mobile Arm SoCs. By implementing strict memory isolation without memory encryption, PORTAL tackles performance, scalability, and power efficiency challenges hindering Arm CCA adoption in the near future. PORTAL leverages CCA's GPC and SMMU for robust hardware-level access control, enabling secure and efficient data interactions between Realm VMs and devices. It also introduces dynamic device management via a dedicated System Realm, facilitating flexible peripheral management at runtime. The evaluation results show that PORTAL improves data transmission and reduces energy consumption. As a pioneering solution, PORTAL lays the groundwork for more practical and widespread deployment of Arm CCA in emerging mobile platforms and other resource-constrained environments.

## REFERENCES

[1] J. Mangalindan, *Is User Data Safe in the Cloud?* http://tech.fortune.cnn.com/2010/09/24/is-user-data-safe-in-the-cloud, Sep. 2010.

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Nov. 2009.

[3] B. W. Lampson, “Lazy and speculative execution in computer systems,” in *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, 2008, pp. 1–2.

[4] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of AES,” in *Cryptographers’ Track at the RSA Conference*, 2006.

[5] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-Level Cache Side-Channel Attacks are Practical,” in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[6] G. Irazoqui, T. Eisenbarth, and B. Sunar, “S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing—and its Application to AES,” in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[7] Y. Yarom and K. Falkner, “Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,” in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[8] J. Van Bulck, F. Piessens, and R. Strackx, “Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic,” in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[9] W. He, W. Zhang, S. Das, and Y. Liu, “SGXlinger: A New Side-Channel Attack Vector Based on Interrupt Latency Against Enclave Execution,” in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, IEEE, 2018, pp. 108–114.

[10] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, “BranchScope: A New Side-Channel Attack on Directional Branch Predictor,” in *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, Mar. 2018.

[11] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazalch, “Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR,” in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, Oct. 2016.

[12] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazalch, “Covert Channels Through Branch Predictors: A Feasibility Study,” in *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2015.

[13] P. Kocher *et al.*, “Spectre Attacks: Exploiting Speculative Execution,” in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[14] M. Lipp *et al.*, “Meltdown: Reading Kernel Memory from User Space,” in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[15] V. Costan and S. Devadas, *Intel SGX Explained*, Cryptology ePrint Archive, Report 2016/086, http://eprint.iacr.org/2016/086.pdf, 2016.

[16] Intel, *Intel Trust Domain Extensions (TDX)*, https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[17] AMD, *AMD Secure Encrypted Virtualization (SEV)*, https://developer.amd.com/sev/.

[18] ARM, “Building a Secure System using TrustZone Technology,” 2009.

[19] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, “Software Grand Exposure: SGX Cache Attacks Are Practical,” in *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, BC, Canada, Aug. 2017.

[20] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “Stealing Intel Secrets from SGX Enclaves via Speculative Execution,” in *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[21] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, “Spectre returns! speculation attacks using the return stack buffer,” in *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT)*, Baltimore, MD, Aug. 2018.

[22] D. Moghimi, J. V. Bulck, N. Heninger, F. Piessens, and B. Sunar, “CopyCat: Controlled Instruction-Level Attacks on Enclaves,” in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[23] J. Van Bulck *et al.*, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[24] J. Van Bulck *et al.*, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[25] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[26] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2017.

[27] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[28] W. Wang *et al.*, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[29] Apple, *Apple Vision Pro*, https://www.apple.com/apple-vision-pro/, 2024.

[30] Qualcomm, *Snapdragon XR2 5G Platform*, https://www.qualcomm.com/products/mobile/snapdragon/xr-vr-ar/snapdragon-xr2-5g-platform, 2024.

[31] Tesla, *FSD Chip*, https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip, 2024.

[32] Apple, *Apple unveils M3, M3 Pro, and M3 Max, the most advanced chips for a personal computer*, https://www.apple.com/newsroom/2023/10/apple-unveils-m3-m3-pro-and-m3-max-the-most-advanced-chips-for-a-personal-computer/, 2023.

[33] Samsug, *Mobile performance redefined*, https://semiconductor.samsung.com/us/processor/mobile-processor/, 2024.

[34] Nvidia, *Nvidia Grace CPU*, https://www.nvidia.com/en-us/data-center/grace-cpu/, 2024.

[35] Nvidia, *Nvidia Jetson Xavier*, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/, 2024.

[36] Qualcomm, *Snapdragon 8 Gen 3 Mobile Platform*, https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-gen-3-mobile-platform, 2024.

[37] ARM, *Introducing Arm Confidential Compute Architecture*, https://developer.arm.com/documentation/den0125/0300/, 2024.

[38] Google, *Confidential Computing*, https://cloud.google.com/security/products/confidential-computing, 2024.

[39] Microsoft, *Azure confidential computing*, https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2024.

[40] AMD, *Confidential Computing Solution Brief*, https://www.amd.com/en/processors/epyc-confidential-computing-cloud, 2024.

[41] G. D. H. Hunt *et al.*, "Confidential computing for OpenPOWER," in *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, Virtual, Apr. 2021.

[42] ARM, *Arm Confidential Compute Architecture*, https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2024.

[43] Linux, *Support for Arm CCA VMs on Linux*, https://lwn.net/Articles/921482/, 2023.

[44] Nvidia, *Confidential Compute on NVIDIA Hopper H100*, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf, 2024.

[45] Arm, *SoC Development*, https://www.arm.com/glossary/soc-development, 2024.

[46] Intel, *Q3 2018 Speculative Execution Side Channel Update*, https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html, 2018.

[47] Intel, *Intel Side Channel Vulnerability MDS*, https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html, 2019.

[48] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[49] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[50] G. Chen *et al.*, "Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[51] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jul. 2018.

[52] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A Commodity Obfuscation Engine on Intel SGX," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[53] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing Your Faults From Telling Your Secrets," in *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May 2016.

[54] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2019.

[55] GCC team, *Using the GNU Compiler Collection (GCC): x86 Built-in Functions*, https://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html, 2019.

[56] Intel, *SGX Tutorial, ISCA 2015*, http://sgxisca.weebly.com/, Portland, OR, Jun. 2015.

[57] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[58] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021.

[59] D. Townley, K. Arıkan, Y. D. Liu, D. Ponomarev, and O. Ergin, "Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[60] G. Dessouky, A. Gruler, P. Mahmoody, A.-R. Sadeghi, and E. Stapf, "Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures," in *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Apr. 2022.

[61] R. Bahmani *et al.*, “CURE: A Security Architecture with CUstomizable and Resilient Enclaves,” in *Proceedings of the 30th USENIX Security Symposium (Security)*, Vancouver, B.C., Canada, Aug. 2021.

[62] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, “Untangle: A Principled Framework to Design Low-Leakage, High-Performance Dynamic Partitioning Schemes,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancouver, Canada, Mar. 2023.

[63] M. Chiappetta, E. Savas, and C. Yilmaz, “Real time detection of cache-based side-channel attacks using hardware performance counters,” *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[64] M. Payer, “HexPADS: A Platform to Detect "Stealth" Attacks,” in *Engineering Secure Software and Systems*, 2016.

[65] J. Chen and G. Venkataramani, “CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Cambridge, UK, Dec. 2014.

[66] M. Yan, Y. Shalabi, and J. Torrellas, “ReplayConfusion: Detecting cache-based covert channel attacks using record and replay,” in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, Oct. 2016.

[67] T. Zhang, Y. Zhang, and R. B. Lee, “Cloudradar: A real-time side-channel attack detection system in clouds,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*, Springer, 2016, pp. 118–140.

[68] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, “Detecting privileged side-channel attacks in shielded execution with Déjá Vu,” in *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Abu Dhabi, UAE, Apr. 2017.

[69] J. Jiang, C. Soriente, and G. O. Karame, “On the Challenges of Detecting Side-Channel Attacks in SGX,” Oct. 2022.

[70] F.-X. Standaert, T. Malkin, and M. Yung, “A Formal Practice-Oriented Model for the Analysis of Side-Channel Attacks,” 2006.

[71] M. S. Taha, M. S. M. Rahim, S. A.-S. Lafta, M. M. Hashim, and H. M. Alzuabidi, “Combination of Steganography and Cryptography: A short Survey,” *IOP Conference Series: Materials Science and Engineering*, vol. 518, 2019.

[72] Arm, *Autonomous Vehicles*, https://www.arm.com/markets/automotive/autonomous-vehicles, 2024.

[73] Nvidia, *In-Vehicle Computing for AI-Defined Cars*, https://www.nvidia.com/en-us/self-driving-cars/in-vehicle-computing/, 2024.

[74] Mobileye, *EyeQ: The System-on-Chip for Automotive Applications*, https://www.mobileye.com/technology/eyeq-chip/, 2024.

[75] Qualcomm, *Snapdragon Ride Platform SoCs*, https://www.qualcomm.com/products/automotive/automated-driving/, 2024.

[76] Y. Ren, J. Li, Z. Yang, P. P. C. Lee, and X. Zhang, "Accelerating Encrypted Deduplication via SGX," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, Jul. 2019.

[77] Y. Deng *et al.*, "StrongBox: A GPU TEE on Arm Endpoints," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, Nov. 2022.

[78] Y. Zhang *et al.*, "SHELTER: Extending arm CCA with isolation in user space," in *Proceedings of the 32st USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[79] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[80] C. Wang *et al.*, "CAGE: Complementing Arm CCA with GPU Extensions," in *Proceedings of the 2024 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2024.

[81] J. Raigoza and K. Jituri, "Evaluating Performance of Symmetric Encryption Algorithms," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2016, pp. 1378–1379.

[82] P. S. Munoz, N. Tran, B. Craig, B. Dezfouli, and Y. Liu, "Analyzing the resource utilization of aes encryption on iot devices," in *2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, 2018, pp. 1200–1207.

[83] A. S. D. Alluhaidan and P. Prabu, "End-to-End Encryption in Resource-Constrained IoT Device," *IEEE Access*, vol. 11, pp. 70 040–70 051, 2023.

[84] P. Porambage, A. Braeken, A. Gurtov, M. Ylianttila, and S. Spinsante, "Secure end-to-end communication for constrained devices in IoT-enabled Ambient Assisted Living systems," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015, pp. 711–714.

[85] I. Sultan and M. T. Banday, "An energy efficient encryption technique for the Internet of Things sensor nodes," *International Journal of Information Technology*, Feb. 2024.

[86] B. Halak, T. Gibson, M. Henley, C.-B. Botea, B. Heath, and S. Khan, "Evaluation of Performance, Energy, and Computation Costs of Quantum-Attack Resilient Encryption Algorithms for Embedded Devices," *IEEE Access*, vol. 12, pp. 8791–8805, 2024.

[87] M. Mössinger, B. Petschkuhn, J. Bauer, R. C. Staudemeyer, M. Wójcik, and H. C. Pöhls, "Towards quantifying the cost of a secure IoT: Overhead and energy consumption of ECC signatures on an ARM-based device," in *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2016, pp. 1–6.

[88] Y. Su and D. C. Ranasinghe, "Leaving Your Things Unattended is No Joke! Memory Bus Snooping and Open Debug Interface Exploits," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2022, pp. 643–648.

[89] S. Ghosh, M. N. I. Khan, A. De, and J.-W. Jang, "Security and privacy threats to on-chip Non-Volatile Memories and countermeasures," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–6.

[90] S. P. Skorobogatov, "Physical Attacks and Tamper Resistance," 2012.

[91] Dayeol Lee and Dongha Jung and Ian T. Fang and Chia-che Tsai and Raluca Ada Popa, "An Off-Chip attack on hardware enclaves via the memory bus," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[92] J. A. Halderman *et al.*, "Lest We Remember: Cold Boot Attacks on Encryption Keys," in *Proceedings of the 17th USENIX Security Symposium (Security)*, San Jose, CA, Jul. 2008.

[93] M. Gruhn and T. Müller, "On the Practicability of Cold Boot Attacks," in *2013 International Conference on Availability, Reliability and Security*, 2013, pp. 390–397.

[94] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors," in *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, Feb. 2017.

[95] J. Mahmod and M. Hicks, "UnTrustZone: Systematic Accelerated Aging to Expose On-chip Secrets," in *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2024.

[96] A. Selinger, K. Rupp, and S. Selberherr, "Evaluation of mobile ARM-based SoCs for high performance computing," in *Proceedings of the 24th High Performance Computing Symposium*, ser. HPC '16, Pasadena, California: Society for Computer Simulation International, 2016, ISBN: 9781510823181.

[97] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-Assisted Secure Execution on ARM Processors," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[98] P. Colp *et al.*, "Protecting Data on Smartphones and Tablets from Memory Attacks," in *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2015.

[99] A. Haas *et al.*, "Bringing the Web up to Speed with WebAssembly," in *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Barcelona, Spain, Jun. 2017.

[100] WebAssembly Community Group, "WebAssembly Specification: Release 1.0," Tech. Rep., May 2019.

[101] H. Wang *et al.*, "Towards Memory Safe Enclave Programming with Rust-SGX," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[102] W. Qiang, Z. Dong, and H. Jin, "Se-Lambda: Securing Privacy-Sensitive Serverless Applications Using SGX Enclave," in *International Conference on Security and Privacy in Communication Systems (SecureComm)*, 2018.

[103] Red Hat, *Enarx*, https://enarx.io, 2019.

[104] Intel, *WebAssembly Micro Runtime*, https://github.com/bytecodealliance/wasm-micro-runtime, 2019.

[105] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "SEISMIC: SEcure in-lined script monitors for interrupting cryptojacks," in *European Symposium on Research in Computer Security*, Springer, 2018, pp. 122–142.

[106] D. Lehmann and M. Pradel, "Wasabi: A Framework for Dynamically Analyzing WebAssembly," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.

[107] *emscripten*, https://emscripten.org/, 2015.

[108] A. Pardoe, *Spectre mitigations in MSVC*, https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, 2018.

[109] J. Seo *et al.*, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[110] *Mirror of the spec testsuite*, https://github.com/WebAssembly/testsuite, 2019.

[111] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[112] Arm, *Fixed Virtual Platforms*, https://www.arm.com/products/development-tools/simulation/fixed-virtual-platforms, 2024.

[113] Orange Pi, *Orange Pi 5 Plus (4GB/8GB/16GB)*, http://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-5-plus.html, 2024.

[114] Intel, *Code Sample: Intel Software Guard Extensions Remote Attestation End-to-End Example*, https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example, 2018.

[115] Intel, *Exception Handling in Intel Software Guard Extensions (Intel SGX) Applications*, 2019.

[116] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, May 2019.

[117] M. Hähnel, W. Cui, and M. Peinado, "High-Resolution Side Channels for Untrusted Operating Systems," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[118] F. Dall *et al.*, "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks," in *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2018.

[119] A. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A false dependency attack against constant-time crypto implementations in SGX," in *Cryptographers' Track at the RSA Conference*, Springer, 2018.

[120] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[121] S. Weiser, R. Spreitzer, and L. Bodner, "Single Trace Attack Against RSA Key Generation in Intel SGX SSL," in *Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Seoul, South Korea, Jun. 2018.

[122] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx, "Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution," in *International Symposium on Engineering Secure Software and Systems*, Springer, 2018, pp. 44–60.

[123] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[124] S. van Schaik *et al.*, "RIDL: Rogue In-flight Data Load," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[125] M. Schwarz *et al.*, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[126] C. Canella *et al.*, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[127] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, "SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017.

[128] O. Ohrimenko *et al.*, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[129] O. Ohrimenko, C. F. Manuel Costa, S. Nowozin, A. Mehta, F. Schuster, and K. Vaswani, “SGX-Enabled Oblivious Machine Learning,” in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[130] S. Sasy, S. Gorbunov, and C. W. Fletcher, “ZeroTrace: Oblivious Memory Primitives from Intel SGX,” in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[131] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, “Obliviate: A Data Oblivious File System for Intel SGX,” in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[132] P. Zhang, C. Song, H. Yin, D. Zou, E. Shi, and H. Jin, “Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks,” in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, Apr. 2020.

[133] J. Lee *et al.*, “Hacking in Darkness: Return-oriented Programming against Secure Enclaves,” in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[134] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, “The Guard’s Dilemma: Efficient Code-Reuse Attacks against Intel SGX,” in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[135] Intel, *3rd Gen Intel Xeon Scalable processors*, https://www.connection.com/~/media/pdfs/brands/i/intel/intel-icelake-ds.pdf?la=en.

[136] O. Acıiçmez and W. Schindler, “A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL,” in *Cryptographers’ Track at the RSA Conference*, Springer, 2008, pp. 256–273.

[137] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX,” in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[138] B. Gras and K. Razavi, “ASLR on the Line: Practical Cache Attacks on the MMU,” in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[139] D. Gruss, C. Maurice, K. Wagner, and S. 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 - Volume 9721*, ser. DIMVA 2016, San Sebastián, Spain: Springer-Verlag, 2016, pp. 279–299, ISBN: 9783319406664.

[140] D. Gruss, R. Spreitzer, and S. Mangard, “Cache template attacks: Automating attacks on inclusive last-level caches,” in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[141] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR,” in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[142] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, “ARMageddon: Cache Attacks on Mobile Devices,” in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[143] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-VM Side Channels and Their Use to Extract Private Keys,” in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[144] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, “Cache attacks on Intel SGX,” in *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[145] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, “TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices,” 2016.

[146] J. Bonneau and I. Mironov, “Cache-Collision Timing Attacks Against AES,” in *Cryptographic Hardware and Embedded Systems (CHES)*, Berlin, Heidelberg: Springer, 2006, pp. 201–215, ISBN: 978-3-540-46561-4.

[147] Intel, *Deprecated Technologies — 12th Generation Intel® Core™ Processors*, [Online; accessed 2-Oct-2022].

[148] Intel, *Performance Monitoring Impact of Intel Transactional Synchronization Extension Memory Ordering Issue*, 2021.

[149] C. Feng, H. Sun, J. Wang, L. Zhang, and S. Xu, “A SoC architecture compatible with Cortex-M4,” in *2023 IEEE 7th Information Technology and Mechatronics Engineering Conference (ITOEC)*, vol. 7, 2023, pp. 514–518.

[150] H. Wang, J. Ma, Y. Yang, M. Gong, and Q. Wang, “A review of system-in-package technologies: Application and reliability of advanced packaging,” *Micromachines*, vol. 14, no. 6, p. 1149, 2023.

[151] Arm, *Layered Security for Your Next SoC*, https://www.arm.com/products/silicon-ip-security, 2024.

[152] A. T. Sheikh, A. Shoker, and P. Esteves-Verissimo, “Resilient and Secure System on Chip with Rejuvenation in the Wake of Persistent Attacks,” in *Proceedings of the 16th European Workshop on System Security*, 2023, pp. 37–43.

[153] M. S. U. I. Sami *et al.*, “Advancing Trustworthiness in System-in-Package: A Novel Root-of-Trust Hardware Security Module for Heterogeneous Integration,” *IEEE Access*, vol. 12, pp. 48 081–48 107, 2024.

[154] D. Mehmedagić, M. R. Fadiheh, J. Müller, A. L. D. Antón, D. Stoffel, and W. Kunz, “Design of Access Control Mechanisms in Systems-on-Chip with Formal Integrity Guarantees,” in *Proceedings of the 32st USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[155] Xilinx, *Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices*, https://docs.amd.com/v/u/en-US/xapp1323-zynq-usp-tamper-resistant-designs, 2024.

[156] Arm, *Arm Architecture Reference Manual for A-profile architecture*, https://developer.arm.com/documentation/ddi0487/latest/, 2024.

[157] Arm, *Arm Realm Management Extension (RME) System Architecture*, https://developer.arm.com/documentation/den0129/latest/, 2024.

[158] Arm, *Arm Mali Graphics Processing Units (GPUs)*, https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus, 2024.

[159] Nvidia, *Tegra X1*, https://developer.nvidia.com/content/tegra-x1/, 2024.

[160] Qualcomm, *Adreno Graphics Processing Units*, https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu/, 2024.

[161] Arm, *Arm System Memory Management Unit Architecture Specification*, https://developer.arm.com/documentation/ihi0070/latest/, 2024.

[162] Intel, *Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation (Revision: 4.1)*, 2018.

[163] Greg, *SGX Attestation results in CONFIGURATION_NEEDED*, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/798777, 2018.

[164] Greg, *GROUP_OUT_OF_DATE - what is the most recent microcode version?* https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/755769, 2018.

[165] Clemens Hammacher, *Liftoff: a new baseline compiler for WebAssembly in V8*, https://v8.dev/blog/liftoff, 2018.

[166] A. Shilov, *Intel's New Core and Xeon W-3175X Processors: Spectre and Meltdown Security Update*, https://www.anandtech.com/show/13450/intels-new-core-and-xeon-w-processors-fixes-for-spectre-meltdown, 2018.

[167] Intel, *Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088*, https://software.intel.com/security-software-guidance/software-guidance/branch-target-injection, 2018.

[168] P. Kocher, *Spectre Mitigations in Microsoft's C/C++ Compiler*, https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[169] *Lighttpd*, https://www.lighttpd.net/, 2003.

[170] *libjpeg*, https://libjpeg.sourceforge.net/, 1991.

[171] *SQLite*, https://www.sqlite.org/index.html, 2000.

[172] *PolyBench*, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2015.

[173] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, "SmashEx: Smashing SGX Enclaves Using Exceptions," in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Seoul, South Korea, Nov. 2021.

[174] Intel, *The latest security information on Intel products*, https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00548.html, 2021.

[175] Open Enclave, *Open Enclave SDK Elevation of Privilege Vulnerability*, https://github.com/openenclave/openenclave/security/advisories/GHSA-mj87-466f-jq42, 2021.

[176] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[177] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys'20, 2020.

[178] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-space Enclaves," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[179] D. Kaplan, “AMD x86 Memory Encryption Technologies,” 2016.

[180] S. Pinto and N. Santos, “Demystifying Arm TrustZone,” *ACM Computing Surveys (CSUR)*, vol. 51, pp. 1–36, 2019.

[181] P. C. Kocher, “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,” in *Annual International Cryptology Conference*, 1996.

[182] I. Biehl, B. Meyer, and V. Müller, “Differential Fault Attacks on Elliptic Curve Cryptosystems,” in *Annual International Cryptology Conference*, 2000.

[183] A. Tang, S. Sethumadhavan, and S. Stolfo, “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management,” in *USENIX Security Symposium*, 2017.

[184] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, “V0LTpwn: Attacking x86 Processor Integrity from Software,” in *USENIX Security Symposium*, 2019.

[185] P. Qiu, D. Wang, Y. Lyu, and G. Qu, “VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies,” *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[186] *Stack frame layout on x86-64*, https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64, 2011.

[187] Y. Ye, R. West, Z. Cheng, and Y. Li, “Coloris: A dynamic cache partitioning system using page coloring,” in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014.

[188] S. Perarnau, M. Tchiboukdjian, and G. Huard, “Controlling Cache Utilization of HPC Applications,” in *International Conference on Supercomputing (ICS)*, 2011.

[189] Y. Yarom, *Mastik: A Micro-Architectural Side-Channel Toolkit*, https://github.com/0xADE1A1DE/Mastik.

[190] T. Kim, M. Peinado, and G. Mainar-Ruiz, “StealthMem: System-level protection against cache-based side channel attacks in the cloud,” in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[191] J. Shi, X. Song, H. Chen, and B. Zang, “Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring,” in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011, pp. 194–199.

[192] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource Management for Isolation Enhanced Cloud Services," ser. CCSW '09, Chicago, Illinois, USA: Association for Computing Machinery, 2009.

[193] D. Gruss, *flush_flush*, https://github.com/IAIK/flush_flush, 2019.

[194] W. Kim, *Fun with Intel Transactional Synchronization Extensions*, https://software.intel.com/en-us/blogs/2013/07/25/fun-with-intel-transactional-synchronization-extensions, 2013.

[195] Pulp-Platform, *CVA6 RISC-V CPU*, https://github.com/openhwgroup/cva6.

[196] Intel, *Intel Xeon Processors*, https://www.intel.com/content/www/us/en/products/details/processors/xeon.html, 2009.

[197] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port Contention for Fun and Profit," May 2019.

[198] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "MicroScope: Enabling Microarchitectural Replay Attacks," Jun. 2019.

[199] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure dynamic cache partitioning for efficient timing channel protection," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[200] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 974–987.

[201] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, San Diego, CA, Jun. 2007.

[202] F. Liu *et al.*, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proceedings of the 22nd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Barcelona, Spain, Feb. 2016.

[203] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[204] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 203–215.

[205] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, “Cache Side-Channel Attacks and Time-Predictability in High-Performance Critical Real-Time Systems,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[206] M. K. Qureshi, “CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 775–787.

[207] M. K. Qureshi, “New Attacks and Defense for Encrypted-Address Cache,” in *Proceedings of the 46th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Phoenix, AZ, Jun. 2019.

[208] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, “Scattercache: Thwarting cache attacks via cache set randomization,” in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[209] Q. Tan, Z. Zeng, K. Bu, and K. Ren, “PhantomCache: Obfuscating Cache Conflicts with Localized Randomization,” in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[210] J. Demme *et al.*, “On the feasibility of online malware detection with performance counters,” *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[211] Z. Allaf, M. Adda, and A. E. Gegov, “A Comparison Study on Flush+Reload and Prime+Probe Attacks on AES Using Machine Learning Approaches,” in *UK Workshop on Computational Intelligence*, 2017.

[212] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapôtre, and G. Gogniat, “NIGHTs-WATCH: a cache-based side-channel intrusion detector using hardware performance counters,” *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2018.

[213] M. Mushtaq, A. Akram, M. K. Bhatti, R. N. B. Rais, V. Lapôtre, and G. Gogniat, “Run-time Detection of Prime + Probe Side-Channel Attack on AES Encryption Algorithm,” *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, pp. 1–5, 2018.

[214] S. Briongos, G. I. Apecechea, P. Malagón, and T. Eisenbarth, “CacheShield: Detecting Cache Attacks through Self-Observation,” *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018.

[215] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Südholt, and J.-M. Menaud, “Cache-based side-channel attacks detection through Intel Cache Monitoring Technology

and Hardware Performance Counters," *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 7–12, 2018.

[216] Y. Kulah, B. Dinçer, C. Yilmaz, and E. Savaş, "SpyDetector: An approach for detecting side-channel attacks at runtime," *International Journal of Information Security*, pp. 1–30, 2019.

[217] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance Counters to Rescue: A Machine Learning based safeguard against Micro-architectural Side-Channel-Attacks," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 564, 2017.

[218] Arm, *Realm Management Monitor specification*, https://developer.arm.com/documentation/den0137/latest/, 2024.

[219] Rockchip, *RK3588 Brief Datasheet*, https://www.rock-chips.com/uploads/pdf/2022.8.26/192/RK3588%20Brief%20Datasheet.pdf, 2022.

[220] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, Ieee, 2009, pp. 44–54.

[221] AlDanial, *cloc*, https://github.com/AlDanial/cloc, 2024.

[222] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[223] Amlogic, Inc., *S905 Datasheet*, https://dn.odroid.com/S905/DataSheet/S905_Public_Datasheet_V1.1.4.pdf, 2016.

[224] Arm, *Juno r2 ARM Development Platform SoC*, https://developer.arm.com/documentation/den0125/0300/Arm-CCA-Hardware-Architecture, 2024.

[225] FuZhou Rockchip Electronics Co., Ltd., *Rockchip RK3288 Technical Reference Manual Part1*, https://opensource.rock-chips.com/images/8/8f/Rockchip_RK3288_TRM_V1.2_Part1-20170321.pdf, 2017.

[226] STMicroelectronics, *GPU device tree configuration*, https://wiki.st.com/stm32mpu/wiki/GPU_device_tree_configuration, 2023.

[227] Synopsys, *Synopsys IDE Security IP Module for PCI Express 5.0*, https://www.synopsys.com/dw/ipdir.php?ds=security-pcie5-ide, 2024.

[228] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *Proceedings of the Twenty-Fourth International*

*Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 455–468.

[229] H. Mai *et al.*, "Honeycomb: Secure and Efficient {GPU} Executions via Static Validation," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 155–172.

[230] J. Zhu *et al.*, "Enabling rack-scale confidential computing using heterogeneous trusted execution environment," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1450–1465.

[231] J. Jiang *et al.*, "CRONUS: Fault-isolated, secure and high-performance heterogeneous computing for trusted execution environment," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 124–143.

[232] H. Park and F. X. Lin, "Safe and Practical GPU Computation in TrustZone," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 505–520.

[233] J. Wang, Y. Wang, and N. Zhang, "Secure and timely gpu execution in cyber-physical systems," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2591–2605.

[234] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 681–696.