RESEARCH-ARTICLE

Interstellar: Fully Partitioned and Efficient Security Monitoring Hardware Near a Processor Core for Protecting Systems against Attacks on Privileged Software

YONGHO SONG, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

BYEONGSU WOO, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

YOUNGKWANG HAN, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

BRENT BYUNGHOON KANG, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

Open Access Support provided by:

Korea Advanced Institute of Science and Technology

Published: 09 December 2024

Citation in BibTeX format

CCS '24: ACM SIGSAC Conference on Computer and Communications Security
October 14 - 18, 2024
UT, Salt Lake City, USA

Conference Sponsors:
SIGSAC

https://doi.org/10.1145/3658644.3690247
ISBN: 9798400706363

Interstellar: Fully Partitioned and Efficient Security Monitoring Hardware Near a Processor Core for Protecting Systems against Attacks on Privileged Software

YongHo Song
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
yonghosong@kaist.ac.kr

Byeongsu Woo
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
wbs79@kaist.ac.kr

Youngkwang Han*
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
sft_glory@kaist.ac.kr

Brent ByungHoon Kang*
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
brentkang@kaist.ac.kr

Abstract

The existing approaches to instruction trace-based security monitoring hardware are dependent on the privileged software, which presents a significant challenge in defending against attacks on privileged software itself. To address this challenge, we propose Interstellar, which introduces a partitioned hardware near the CPU's main core and leverages the benefit of hardware-level security monitoring. Interstellar is fully partitioned, parallelized, and simultaneously detecting security monitoring hardware. Interstellar's design makes it hard for malicious software to reverse-engineer how Interstellar detects the attacks, and Interstellar efficiently protects the system against the attacks on the privileged software (e.g., *Trusted Execution Environment (TEE)*). Moreover, Interstellar not only monitors but also blocks various attacks in a timely manner without stalling a CPU core by designing with a finite-state machine.

We implemented a prototype of Interstellar in Rocket chip using a hardware description language and evaluated Interstellar with a Linux kernel and a custom TEE-equipped Linux kernel for Rocket chip on two different FPGA boards. The performance overhead of Interstellar is negligible for benchmark applications. The average performance overhead incurred from Interstellar on 50MHz Rocket core for three different benchmarks is 0.102%.

CCS Concepts

• **Security and privacy → Hardware security implementation**; *Operating systems security*; Information flow control.

Keywords

Hardware security; Partitioned security monitor; Instruction-tracing; Finite-state machine

*B. Kang and Y. Han are the corresponding authors.

Y.H. Song, B. Woo, Y. Han, and B. Kang are also affiliated with CySecuLab.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690247

ACM Reference Format:
YongHo Song, Byeongsu Woo, Youngkwang Han, and Brent ByungHoon Kang. 2024. Interstellar: Fully Partitioned and Efficient Security Monitoring Hardware Near a Processor Core for Protecting Systems against Attacks on Privileged Software. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690247

1 Introduction

Progress in privilege-based systems. Security mechanisms for modern computing systems are dominantly based on granting different privileges to each software in a hierarchical manner, and the system software that is protected by a privilege mechanism is employed to prevent faults and malicious behaviors in some applications from propagating to their entire system. For example, an *Operating System (OS)* kernel can access privileged resources with its privileged instructions, which can only be executed in the kernel mode of the CPU hardware. Also, the OS kernel can manage the page mapping for applications by leveraging the OS privilege.

However, the system software, such as the OS kernel, has a probability of finding vulnerabilities in the system software due to its large attack surface. Also, the system software can access any classified data related to the system's security or private data using its privilege. Thus, if an attacker uncovers vulnerabilities in a large amount of code of the system software, the attacker can easily leak important data by exploiting the vulnerabilities and by taking advantage of the system software's privilege. To protect important data from the system software that has large attack surfaces, *Trusted Execution Environments (TEEs)* [26, 48] and custom TEEs [24, 27, 39, 52] have been proposed on the privilege-based systems by introducing additional minimal *Trusted Computing Base (TCB)*. Still, the trusted software and security hardware components included in TCB depend on another privilege-based security mechanism.

Security vulnerabilities in the design of privilege-based systems. Unfortunately, despite these advances in the privilege-based systems, the existing attacks on OS kernel that exploit vulnerabilities in the privileged software's code and shared hardware to leak secrets are still effective on the TCB of the TEE or custom TEE with some variations. For example, a *Return-oriented Programming (ROP)* attack [23, 40], one of the attacks that work effectively on both OS and TEE if the attacker finds any buffer overflow vulnerability and necessary gadgets among the code of the privileged software

or the application executed on TEE. In addition, hardware side-channel-based attacks [20, 21, 41, 42], which exploit vulnerabilities in microarchitectural hardware resulting from design flaws or interference within shared hardware between different privileges, are also still effective not only on the OS kernel, but also on the TEEs and many custom TEEs.

Interestingly, the existing privilege-based system, including the OS kernel and TEE, has a generalized design principle for implementing its security mechanism. They have TCB consisting of trusted software, hardware security primitives, and dedicated microarchitectural components. Then, they utilize the trusted software that has the highest (or special) privilege to isolate the TCB from the other vulnerable software. However, this generalized design still faces difficulties in that the privileged software included in the TCB must be free from any vulnerabilities that software-based attacks can exploit. In addition, the hardware security primitives and the dedicated microarchitectural components included in the TCB must be safe from hardware side-channel attacks across different privileges.

Drawbacks of prior related works. Prior works [30, 31, 33] for instruction tracing security monitoring hardware get the instructions executed by the monitored software and their corresponding microarchitectural information to detect malicious actions or accelerate security features for applications running in privilege-based systems. However, all the prior security monitoring hardware is directly controlled by the privileged software (i.e., OS kernel), same with the design of the security mechanism for the privilege-based system, to support the software-programmability on security monitoring rule utilized by the monitoring hardware. Also, some prior works [30, 31] do not include the cache side-channel attacks in their threat models. Hence, the prior works are also vulnerable to existing attacks on the privileged software, and the prior works cannot be utilized for protecting the TEE against OS-level attackers.

Moreover, due to their hardware implementation to support the software-programmability on the security monitoring, some prior works [30, 31] cannot monitor the software running on the main core simultaneously, so it is difficult to detect attacks before attacks are carried out. Also, another work [33] is unable to generate comprehensive security monitoring rules by referring to various microarchitectural resources. To sum up, these design choices of prior works for the instruction tracing security monitoring hardware cannot leverage the benefits of security monitoring hardware at the microarchitecture level, which can safely and efficiently detect attacks across different privileges.

Interstellar. To efficiently protect the privilege-based systems against existing attacks on privileged software, this paper presents Interstellar, which is fully partitioned, parallelized, and simultaneously detecting instruction tracing security monitoring hardware. Interstellar addresses the limitations of prior related works [30, 31, 33] and leverages the benefits that security monitoring hardware can achieve at the microarchitecture level.

In particular, ① Interstellar introduces a design of fully partitioned security monitoring hardware, which is not accessible from any software and is separated from the CPU's main core and cache, to safely detect and block the existing attacks on privileged software. In addition, ② Interstellar monitors every fetched instruction from software running on a CPU main core in parallel using multiple attack detection rules. Interstellar utilizes *Finite-state Machines (FSMs)* to efficiently implement multiple comprehensive attack detection rules, which refer to the executed instructions and the various corresponding information at the microarchitecture level for the attack detection. Lastly, ③ Interstellar can simultaneously detect attacks performed on the main core before the attacks are carried out (i.e. before the instructions for the attacks are committed). Notably, since Interstellar can be optimized at the microarchitecture level when designed for each attack detection case, Interstellar can block the detected attacks in a timely manner without stalling the main core's pipeline.

Furthermore, to enable bug-free simultaneous detection on Interstellar coupled with the pipeline core of the Rocket chip [15], we address the following challenges. First, FSM of Interstellar must be designed considering the instruction squashing situation resulting from the branch misprediction or hardware interrupt before the monitored instructions are committed. We address this problem by introducing an instruction squashing handler for the correct recovery of the FSM's state from the invalidation situation. Second, to avoid stalling the progress of the main core, Interstellar must determine attack detection results by referring to microarchitectural information within at most three CPU clock cycles between the fetch stage and commit stage of the Rocket chip's main core. To solve this challenge, we optimize the design of the FSM to directly refer to the required microarchitectural information in each pipeline stage for each attack detection use case. In addition, we design the determination logic in the FSM to complete the detection of the attack within one CPU clock cycle.

To evaluate Interstellar with three attack monitoring use-cases implemented in parallel, we implement a prototype of Interstellar in a RISC-V Rocket chip using Chisel [17], a hardware description language. We evaluated the Interstellar-enabled Rocket chip on the AMD Vertex 7 FPGA VC707 board [6] with TEE-enabled Linux kernel and on AWS EC2 F1 utilizing Firesim [8] with RISC-V Linux kernel. Also, we verify the functionalities of Interstellar and analyze that Interstellar is safe from existing and possible threats. Our FPGA-based evaluation demonstrates that the performance overhead of Interstellar is negligible on average when executing three benchmarks [9–11]. Meanwhile, the area overhead and relative power consumption of Interstellar with all three attack detection rules compared to the Rocket core are 21.72% and 34.10%, respectively.

Our contribution. In summary, these are our contributions:

- To protect privilege-based systems from existing attacks on privileged software, we present Interstellar, a safe security monitoring hardware that can be utilized for protecting OS kernel and TEE, by introducing fully partitioned monitoring hardware.
- To achieve both the efficient parallel monitoring with multiple attack detection rules and the simultaneous detection for blocking the attacks in a timely manner, Interstellar utilizes FSMs to define and optimize each attack detection rule in hardware logic, considering the behavior of pipelined Rocket core.
- We evaluate a prototype of Interstellar implemented in Rocket chip with TEE-enabled Linux kernel and RISC-V Linux kernel on two different FPGA boards by running three different benchmarks, and the performance overhead of 50MHz Rocket core for three different benchmarks is 0.102%, on average.

2 Background

2.1 Designs of the Systems Protected by Privilege-based Mechanisms

There are the following well-known privilege mechanisms for the security of systems: (1) a protection ring and (2) *Trusted Execution Environments (TEEs)*. In addition, various (3) custom TEEs have been proposed to address security problems in existing TEEs.

Trusted Execution Environments and Protection Ring. A protection ring is a representative privilege-based security mechanism that hierarchically protects core functionalities of computer systems from malicious software. To realize the protection ring in computer systems, the CPU employs dedicated hardware components, such as descriptor tables, dedicated registers, and privileged instruction. By leveraging the hardware components designed for the ring protection, the processor provides segment-level access control and a call gate that facilitates the safe transfer of the program execution between different privilege levels. Moreover, the system protected by the protection ring employs privileged system software to realize page-level protection. The system software can utilize the protection ring's hardware components and privileged instructions to protect the code and data of the system and applications in a fine-grained unit.

However, prior attacks [19, 23, 29, 44] can leak important data in the privileged system software, although the systems are protected by the protection ring. The attacks bypass the privilege check procedure of the protection ring by exploiting vulnerabilities in the privileged software and in the microarchitectural design. More importantly, the security threat to the privilege-based systems that exploit vulnerabilities in the privileged system software is an ongoing problem and can occur at any time, given the massive amount and open-sourced privileged system software.

To make a secure execution environment isolated from the large system protected only by the protection ring, CPU vendors (e.g., Intel and ARM) have proposed TEEs [2, 26, 48]. TEE provides an isolated execution environment in systems, even in situations where the privileged system software (e.g., *Operating System (OS)*) is compromised by attackers. TEE utilizes a new hardware-driven privilege-checking mechanism with additional privilege mode (e.g., *Software Guard Extension (SGX)* enclave mode) and dedicated hardware components. Then, CPU vendors employ new small-sized privileged software for the applications run in TEE, considering small-privileged software and dedicated hardware components as minimal *Trusted Computing Base (TCB)*, as shown in Figure 1. Then, the CPU vendors implement the security services of the TEE for applications, such as the memory access control and the safe transfer of the program execution between the normal and TEE modes, by utilizing the privilege mechanism for TEE and the privileged instructions for the software in TCB.

However, despite efforts to construct safe TEEs on the privilege-based system, many prior studies [18, 21, 40–42, 46, 47] have demonstrated that attacks to leak secrets in the TEEs from lower-privileged software are feasible. Especially, most prior attacks on TEEs were performed using similar attack methods that were also valid against OS. In general, prior studies have used three methods in order to leak secret data in the TEEs: (1) Privileged hardware resources abusing attacks [46, 47], (2) *Return-oriented Programming (ROP)*-based attacks [18, 40], and (3) Hardware side-channel attacks [21, 41, 42]. In Table 1, we categorize the previous studies for attacks on privileged software according to the types and directions of the attacks.

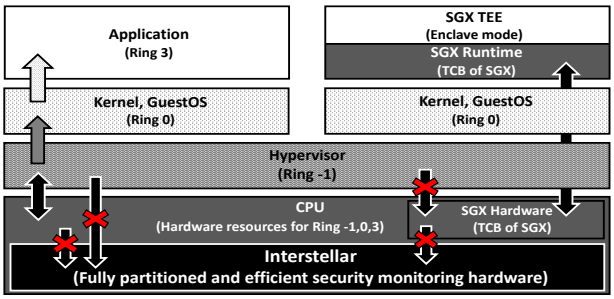

Figure 1: Security design for the privilege-based system using the protection ring and SGX, and the position of Interstellar. Each arrow means permitted or prohibited access direction.

Custom trusted execution environments. To address the security problems in the TEEs, prior studies [24, 27, 34, 39, 52] have proposed the design of a custom TEE by extending the existing hardware components or security primitives and by again introducing new highest-privileged software that can utilize the extended security hardware. Thus, these prior studies are still unable to guarantee the security of their design unless they can ensure that their highest-privileged software is free of vulnerabilities.

First, Sanctum [27] minimally modified hardware and implemented security monitoring software with the new highest machine privilege to protect SGX enclaves against the side-channel attacks such as cache timing attacks [20] and memory access pattern attacks [51] from lower-privileged software. For example, to mitigate memory access pattern attacks on SGX enclaves, Sanctum implemented a page-coloring-based cache partitioning scheme by adding a new cache address shift hardware, modifying the existing page table walker for each enclave, and introducing the machine-privileged security monitoring software.

Next, Keystone [39] provides an open framework that enables CPU architects and TEE programmers to create custom TEEs on existing RISC-V systems. Also, Keystone realizes safer custom TEE, which also can mitigate cache-side channel attacks, on the RISC-V CPU by introducing machine-privileged security monitoring software and utilizing RISC-V's hardware primitives, such as *Physical Memory Protections (PMPs)*.

Subsequently, Penglai [34] proposes a scalable TEE on RISC-V, overcoming scalability limitations of legacy TEEs, such as Intel SGX [26], in terms of memory isolation and integrity protection. Penglai addressed these limitations by implementing hardware primitives, *Guarded Page Table (GPT)* and *Mountable Merkle Tree (MMT)*. Furthermore, Penglai introduced the Secure Monitor, a security monitoring software that operates in the most privileged mode and is responsible for the management of enclaves and protection against the manipulation of hardware primitives.

Lastly, ReZone [24] is a custom TEE designed to address the problem of ARM Trustzone's environment, which is easy to take control of the OS in the normal world, trusted application, and secure monitor if the trusted OS inside the secure world is compromised. ReZone introduces secure monitoring software with EL3 privilege and utilizes Trustzone-agnostic hardware primitives available on

Types of Attacks	Prior Attacks	Exploited Vulnerabilities	Attack Methods	Direction of Attacks
Privileged hardware components abusing attacks	Plundervolt [46]	Legitimate privileged dynamic voltage scaling interfaces for privileged software	Abusing MSRs, which control the dynamic voltage scaling from software, to inject faults into enclave by decreasing CPU voltage	OS software / BIOS ⇒ SGX enclaves (TEE)
	PMP poisoning [47]	Rocket chip's main core skips the instruction fetch if an attacker injects a short glitch into the device's clock	Corrupting PMPs, which control the memory access permissions, by bypassing the fetch of PMP configuration instructions	Application ⇒ Rocket chip's TEE
Code reuse or Control flow abusing attacks	ROP is still dangerous [23]	Common features of prior ROP defenses that can be circumvented (Length-based classifiers or Monitoring a limited amount of history)	Bypassing prior defenses with mixing normal-length gadgets or adding no-op instructions (i.e., Ad hoc approaches to disclosed defenses)	Application ⇒ Linux / Windows kernel
	DarkROP [40]	Buffer overflow vulnerabilities in encrypted enclave program and exception handling mechanism of SGX	Finding gadgets and associated exploits in an enclave by using enclave exception handler and memcpy	OS + Host application ⇒ SGX enclaves
Hardware side-channel attacks	Foreshadow [21]	Speculative execution vulnerability to enclave's data loaded in L1 data cache in Intel x86 CPU, similar to Meltdown Attack [43]	Leaking plaintext secret of enclave prefetched on L1 data cache with speculative execution and FLUSH+RELOAD cache covert channel	Application or Linux kernel ⇒ SGX enclaves
	Branch shadowing [41]	SGX doesn't clear branch history in the shared hardware (BTB, BPU, and LBR) when switching from enclave to non-enclave mode software	Obtaining control flow of an enclave by writing shadow code that probes the history of the enclave's branches	OS kernel ⇒ SGX enclaves

Table 1: Classification of some prior attacks on privileged software (MSRs: Model-Specific Registers, ROP: Return-Oriented Programming, BTB: Branch Target Buffer, BPU: Branch Prediction Unit, LBR: Last Branch Record)

general commercial hardware platforms to divide the monolithic TEE into multiple sandboxes to disarm the trusted OS's privilege.

Security implications for privilege-based system design. In general, the existing system design to enhance the security of privilege-based systems *introduce additional privileged instructions and dedicated security hardware components (or primitives) within the system that operate only with new (or highest) privilege*. Then, in order to isolate a new secure execution environment from the unsafe system, the existing methods *appoint new privileged software that can take advantage of the security hardware features through the new privileged instructions*. The generalized system design has been repeatedly applied when TEEs is introduced within the protection ring and when creating safer custom TEE for the existing TEE.

Meanwhile, some prior attacks on OS kernel or TEE [18, 23, 40] leak sensitive data by exploiting vulnerabilities in higher-privileged (or trusted) software with control flow hijacking or code reuse methods. Other prior attacks [21, 42] exploit hardware vulnerabilities resulting from interference in shared hardware or flaw in microarchitectural design by using side-channel attack methods. Thus, unless the generalized design of the privilege-based systems can thoroughly eliminate the vulnerabilities in the introduced higher-privileged software or the interference in the microarchitecture hardware shared by software across different privileges, prior attack methods will continue to be exploited with some variations to defeat the privilege-based systems.

2.2 Prior Works for Instruction Tracing Security Monitoring Hardware

Prior works for instruction tracing security monitoring hardware [30, 31, 33] add a new security monitoring co-processor connected with the main core of Rocket chip [15] by minimally modifying the main core's microarchitecture. The security monitoring hardware constantly traces both the program's instructions executed on the main core and the related microarchitectural states of the main core during runtime to provide monitoring services for the system's security. Then, the security monitoring hardware provides the security service based on the software-programmed monitoring rules that can be defined with the privileged instructions introduced for software users. The comparison between the related works and Interstellar are summarized in Table 2.

Prior related works. Nile [31] is the first to propose programmable monitoring hardware (i.e., a coprocessor) that can trace and utilize all executed instructions and the related microarchitectural information to detect the occurrence of software user-defined events. Existing hardware performance counters [28] can be used as a performance analysis and debugging tool, but they have limitations when used to monitor events such as security verification. To overcome their limitations, Nile proposes hardware that can monitor events using the architectural state of the CPU core per executed instruction. Then, Nile provides interface functions, which include privileged instructions that can control the monitoring hardware, for software-level users to program the security events that they want to monitor with the coprocessor. Nile programmed a shadow stack [22] on its monitoring coprocessor for a use case to detect ROP attacks.

Then, PHMon [30] improves Nile's coprocessor to support not only the simple comparison operation but also arithmetic or logical operations to program other security use cases, such as accelerating fuzzing or preventing the leakage of sensitive information stored at a specific memory address. For evaluation, PHMon modifies a RISC-V rocket chip to implement instruction tracing security monitoring hardware on an FPGA board. PHMon demonstrates that the monitoring hardware causes little performance overhead and can perform security monitoring more efficiently than the existing security monitoring software for the same purpose.

Lastly, ISA-grid [33] designs an instruction tracing security monitoring hardware that enables fine-grained monitoring for the access violations to *Control and Status Registers (CSRs)*, which store important data necessary for hardware management and security services. ISA-grid provides new privileged instructions for software developers to create domains that have their own privileges, called domain ID, beyond the kernel privileges. The ISA-grid's privileged instructions are utilized to create decoupled domains with additional privileges and specify access-controlled CSRs within each domain. By using privileged instructions, users can ensure that the attacker cannot access sensitive data stored in other domains' CSRs,

	Dedicated security monitoring hardware	Various microarchitectural info. can be utilized for monitoring	Trusted privileged software is safe from OS-level attackers	Parallelized monitoring with multiple security rules	Simultaneous detection	Performance overhead
Nile [31]	✓	✗	✗	✗	✗	Medium
PHMon [30]	✓	✓	✗	✓	✗	Medium
ISA-Grid [33]	✓	✗	✗	✗	✓	Low
Interstellar	✔	✔	**N/A**	✔	✔	**Very Low**

Table 2: Comparison table between the related works on instruction tracing security monitoring hardware and Interstellar

which are required for critical hardware management or related to security services, even if an attacker gains kernel privileges.

Drawbacks of prior works. Nile [31] first proposed the concept of a programmable security monitoring coprocessor. However, despite presenting the shadow stack as its security use-case, Nile allows both application-level and OS-level users to directly program and control the monitoring coprocessor with custom instructions through the interfaced functions of Nile, without any explanation of the threat model. Moreover, even the application-level user can disable the monitoring hardware through the interfaced function. Therefore, Nile's programming for the shadow stack on their monitoring coprocessor would be impractical in privilege-based systems.

Next, PHMon [30] clarifies its threat model assuming that PHMon trusts OS kernel, unlike Nile. However, PHMon is still unable to be utilized in a threat model where the attacker can take advantage of the OS privilege in the privilege-based systems because PHMon trusts the OS kernel to support software programmability for the monitoring coprocessor and to manage the security monitoring hardware at the OS level.

Moreover, PHmon has limitations in its security capability due to its decoupled monitoring for non-blocking execution of the main core; otherwise, it incurs significant performance overhead. The reason is that PHMon designed the security monitoring coprocessor to collect all the instructions and microarchitectural information necessary for detecting security violations only at the commit stage of the main core in the form of committed logs. In this design, if the main core must stall to receive the security detection result from PHMon every time after sending the committed log at the write-back stage, it will affect the performance of the CPU main core (i.e., PHMon cannot support simultaneous detection). To address this performance issue, PHMon decouples the security monitoring from the execution of the main core in a non-blocking manner. In the end, because of the decoupled monitoring, PHMon cannot block the commit of malicious instructions on the main core until the monitoring hardware detects the security violation by referring to the committed logs that are sent at the main core's commit stage.

In addition, PHMon's security monitoring configuration would be vulnerable to cache side-channel attacks because the user-programmed configurations necessary for PHMon's security monitoring are sent from the main memory via the CPU's data cache to the monitoring hardware. Furthermore, PHMon must compile the software to be monitored by the coprocessor using the custom compiler implemented by PHMon to collect signals that are included in the commit log from the main core at each stage.

ISA-grid [33] aims to enable developers to decouple CSRs, which contain important information necessary for OS's security services, from vulnerable OS kernel by creating domains with additional unique privileges (i.e. domain ID) according to the types of CSR's data. However, although ISA-grid allows developers to isolate CSRs from vulnerable OS through domain-0 software that manages the entire domain to protect CSRs with the highest privilege, ISA-grid does not present the threat model it assumes for domain-0 software.

Without the threat model, it is impossible to explain how the domain-0 software is safe against attacks that exploit software vulnerabilities of privileged software to bypass the privilege check mechanism, such as ROP attacks. While no threat model is presented, ISA-grid seems to assume that a supervisor-level developer can create and control domains with gate instructions introduced by ISA-grid via domain-0 software; thus, ISA-grid also cannot be utilized in privilege-based systems where the TEEs exists because TEE's threat model does not trust supervisor-level software.

Lastly, ISA-grid only provides security monitoring capabilities related to CSRs, making it difficult for developers to implement comprehensive security use cases, such as detecting existing attack methods on privileged software, such as control flow hijacking, code reuse attacks, and cache side-channel attacks.

3 Threat Model and Assumption

This work focuses on safe detection and blocking of various attacks on higher-privileged software in privilege-based systems. Attackers can use the functionalities of low-privileged software to leak secret data of higher-privileged software by exploiting vulnerabilities in the higher-privileged software or in the microarchitecture components shared across software with different privileges.

To safely detect attacks on higher-privileged software against the given attacker's capabilities, Interstellar trusts only the initial setup procedure for higher-privileged software at boot time. We assume that the boot process's trustworthiness is ensured by secure boot technology. For each attack scenario, after the initial setup at the booting, Interstellar does not trust the lower-privileged software that the attacker can take advantage of the privilege.

Hence, Interstellar can be used for monitoring the attacks on the OS and TEE of the lower privileged software. In this work, we utilize Interstellar to detect these attacks on privileged software: unauthorized memory access to TEE, ROP attacks on TEE, and microarchitectural timing side-channel attacks on the data of higher-privileged software.

Meanwhile, we assume that the CPU's microarchitectural hardware operates normally as designed (i.e., bug-free). In addition, other than the vulnerabilities in the higher-privileged software in each attack scenario, the code and data of the software are executed correctly. For example, if Interstellar is utilized for detecting the ROP attacks on TEE with OS privilege, both the application code in TEE that is not manipulated by the attackers and the TEE's dedicated hardware work normally as implemented. Lastly, we assume that Interstellar is safe from the physical thermal and power

side-channel attacks [36, 45] that utilize the external analysis instruments in a situation where physical access is possible.

4 Interstellar

We propose a fully partitioned, parallelized, and simultaneously detecting security monitoring hardware (i.e., Interstellar) near a processor core to protect systems against attacks on privileged software.

4.1 Objectives of Interstellar

High security. To make secure instruction tracing security monitoring hardware, Interstellar defines attack detection rules using fully partitioned hardware logic, which is not accessible with any privileged instructions by the privileged software. In addition, to partition Interstellar also from the CPU's main core, Interstellar filters the incoming fetched instructions from the main core using an instruction hardware filter that works based on hardware logic-defined attack detection rules. The instruction filter ensures that only permitted information associated with monitored instructions can be passed into fully partitioned Interstellar to prevent the intrusion of malicious instructions and data, as shown in Figure 2. Lastly, Interstellar has its own separate SRAM scratchpads, so the microarchitectural information required for Interstellar cannot be inferred from the existing hardware-side channel attacks, such as cache side-channel attacks.

Parallelized attack monitoring. To achieve parallelized monitoring for multiple attack detection rules for every incoming fetched instruction, the attack detection rules of Interstellar are implemented with multiple sets of hardware logic. Furthermore, each attack detection rule is implemented efficiently with a simple *Finite-state Machine (FSM)* to minimize the area overhead of Interstellar. The monitored instruction and the required microarchitectural information are sent through the instruction filter to each FSM that detects a target attack. Then, each FSM determines that the attacks have been carried out with separated states in parallel by referring to a set of the instructions and their associated information.

Simultaneous attack detection. To simultaneously detect a targeted attack before the attack is committed on a main core without stalling the main core, we optimize FSM for each attack detection rule at the microarchitecture level, considering the referred microarchitectural information by each monitoring rule and the pipeline structure of the main core.

In a high-level description, Interstellar receives fetched instructions of monitored software at the beginning of the decode stage of the CPU's main core. Then, Interstellar performs security monitoring based on the attack detection rules while the monitored instruction is being executed on the main core. Thanks to the FSM-based design, the security monitoring logic of Interstellar can determine whether the monitored instructions are intended for malicious behavior before the instruction is committed in the write-back stage of the main core by referring to the associated microarchitectural information and trusted security configuration. Based on the result of the attack detection, Interstellar can block attacks at the microarchitecture level in a timely and appropriate manner for each use case. This is because, unlike prior studies [30, 33], Interstellar does not program security monitoring rules in the software for a security monitoring processor, which is generally designed for

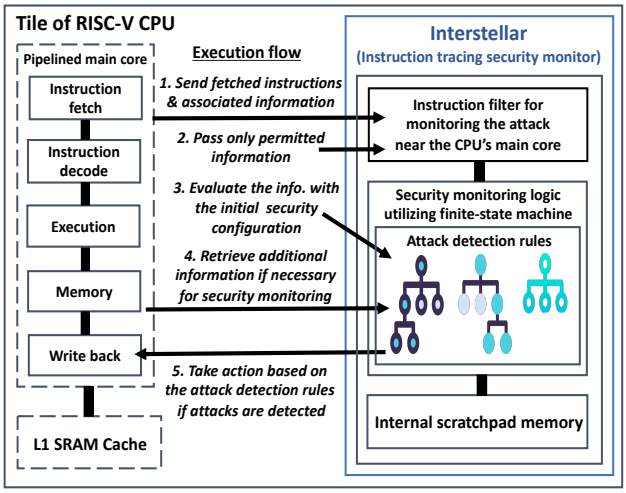

Figure 2: High-level microarchitecture design of Interstellar near the main core and execution flow

software programmability. In the end, Interstellar can address the limitations of the prior works that are unable to block the attack before performed and hard to be optimized at the microarchitecture level, which are explained in §2.2.

4.2 Design of Interstellar

Instruction filter. The instruction filter determines which instructions and associated information should be collected from every fetched instruction on the main core to detect attacks, as described in Figure 2. Since the instruction filter is designed with the same *Instruction Set Architecture (ISA)* of the main core, the instruction filter can decode the type and opcode of the instruction by referring to the main core's opcode table. The filtering list of the instruction filter keeps changing depending on the attack detection rules defined by FSMs inside the security monitoring logic. Specifically, if the passed instruction and information transit the state of the FSM, the filtering list inside the instruction filter may change. Interstellar is operated near the CPU's main core and receives the fetched instructions from the main core using our communication protocol. It is important to note that Interstellar does not inform the result of the instruction filtering to the main core to prevent inferring the attack detection rule from the filtering result.

Security monitoring logic. A security monitoring logic of Interstellar detects malicious software's actions that perform the prior attacks on privileged software from the CPU's main core based on attack detection rules. The security monitoring logic consists of multiple attack detection rules, and each attack detection rule is defined with a separate FSM that runs in parallel.

The FSM-defined rule detects malicious software's actions by evaluating the fetched sequence of monitored instructions and the main core's microarchitectural information associated with the monitored instructions. For attack detection, FSM has sets of registers for storing both the trusted initial security configuration and the microarchitectural information that is passed through the instruction filter or is retrieved after the main core's fetch stage, as described in Figure 3. Additionally, the states of FSM have their own register sets to preserve the necessary information of each state even after the state transits, as shown in Figure 3. This is because

FSM needs to restore the previous state if the monitored instruction simultaneously executed on the main core is squashed because of a branch misprediction or a hardware interrupt. Notably, we also implement an instruction squashing handler with FSM, which will be explained in subsection 5.1, to restore the previous state when a branch misprediction or interrupt occurs.

In addition, FSMs are responsible for updating the filtering list in the instruction filter. Since Interstellar monitors the sequence of fetched instructions to detect attacks, the filtering list in the instruction filter must be updated depending on the state transition in FSMs. Therefore, when a state transition occurs, the security monitoring logic promptly notifies the instruction filter to update the filtering list.

Internal scratchpad memory. Interstellar utilizes dedicated internal scratchpad memory to repeatedly store the same set of microarchitectural information for long-term monitoring that is necessary for detecting a particular attack because the FSM cannot support the long-term monitoring only with a set of registers. For example, each FSM of Interstellar can utilize its own internal scratchpad memory like a stack for identical long-term attack detection using a dedicated stack pointer. Interstellar stores the required microarchitectural information stored in the state's register sets into the scratchpad memory if both conditions are met: ① the state has a self-loop for identical monitoring; ② the monitored instruction is committed at the main core's write-back stage, and the associated microarchitectural information will be utilized later for long-term monitoring.

More importantly, the internal scratchpad is only accessible by the associated FSM using the hard-wired address in each FSM so that the software-level attacker cannot access the microarchitectural information stored in the scratchpad memory of Interstellar and also cannot leak the information stored in the scratchpad memory by performing existing side-channel attacks on the shared data cache. Also, we employ the scratchpad memory design for the internal memory rather than the SRAM cache design attached to the main core because the microarchitecture structure and the communication method of the internal memory must be simple and efficient to support simultaneous detection.

Communication protocol. Interstellar communicates with the CPU's main core with our communication protocol, which is designed upon the ready/valid signal handshake protocol [4, 5] at the microarchitecture level to accurately and efficiently receive and retrieve the necessary information from the main core. As shown in Figure 2, Interstellar mainly receives instruction and the associated information from the instruction fetch stage. However, some required information can be retrieved after the fetch stage, such as a branch misprediction flag, with Interstellar's communication protocol. In addition, each step in Interstellar's communication protocol is completed in a single cycle to achieve simultaneous detection without stalling the main core.

In addition, unlike prior work [30], Interstellar does not utilize a Rocket Custom Coprocessor (RoCC) interface to communicate with the main core for two reasons. First, Interstellar cannot achieve simultaneous detection by using the RoCC interface. This is because, when using the RoCC interface, the commands for the coprocessor are generated by the RoCC instruction committed on the main core, which means that the coprocessor can receive the information required for monitoring only after the main core's commit stage. Secondly, the RoCC interface is unnecessarily complicated for Interstellar because Interstellar does not need the RoCC features, such as communication with the L1 data cache and RoCC instruction for software, and these features may even compromise the security of Interstellar, which is designed against attacks on privileged software.

4.3 Execution Flow of Interstellar

Initial setup. During the system's boot process, Interstellar establishes the necessary initial security configurations for each attack detection rule. Since the boot process of the system is trusted by Interstellar, the initial security configuration information is trusted security information that can be leveraged by attack detection rules to detect attacks on privileged software. Initial security configurations are categorized into two types: invariant configurations, which are independent of the system and machine, and variant configurations, which depend on the system and machine.

For the first type, initial security configurations are hard-wired according to each attack detection rule and integrated into Interstellar independently of the system's boot process. However, for the second type, Interstellar monitors the system's boot process to obtain and set necessary initial security configurations. For example, although Interstellar requires the information of RISC-V PMP as the initial security configuration to detect attacks, the PMP information cannot be hard-wired in Interstellar because the protected physical address boundaries vary for each machine. In this case, to obtain the required initial configuration, Interstellar monitors specific instructions that are associated with the configuration during the boot process, such as RISC-V's CSRRW (Atomic Read and Write CSR) instruction for setting PMP information. Then, the initial security configurations are stored and kept only in the dedicated registers of each attack detection rule inside of Interstellar, preventing the initial configuration from being compromised by malicious access after the boot process.

Instruction filtering. Once the initial setup for each monitoring rule is done, Interstellar utilizes an instruction filter to filter incoming instructions and the required microarchitectural information. First, Interstellar identifies the monitored instructions among the fetched instructions from the main core referring the current filtering list and an opcode table in Interstellar. Then, Interstellar filters the necessary information (e.g., program counter and mode bit) associated with the monitored instruction based on the filtering lists to determine whether the information is necessary for each attack detection rule. If the newly fetched instruction is not one of the monitored instructions, the instruction filter drops the transferred instruction.

In addition, the communication protocol between Interstellar and the main core always checks the availability of the information transfer to each other using the ready/valid handshake protocol before the actual information (i.e., data) is transferred. After the instruction fetch stage of the main core, the fetched instruction and the associated information are transferred to the instruction filter of Interstellar when both ready/valid bits are set to 1.

Security monitoring. The microarchitectural information permitted by the instruction filter is transmitted to each FSM that requires the information to detect the attacks. The FSM stores the

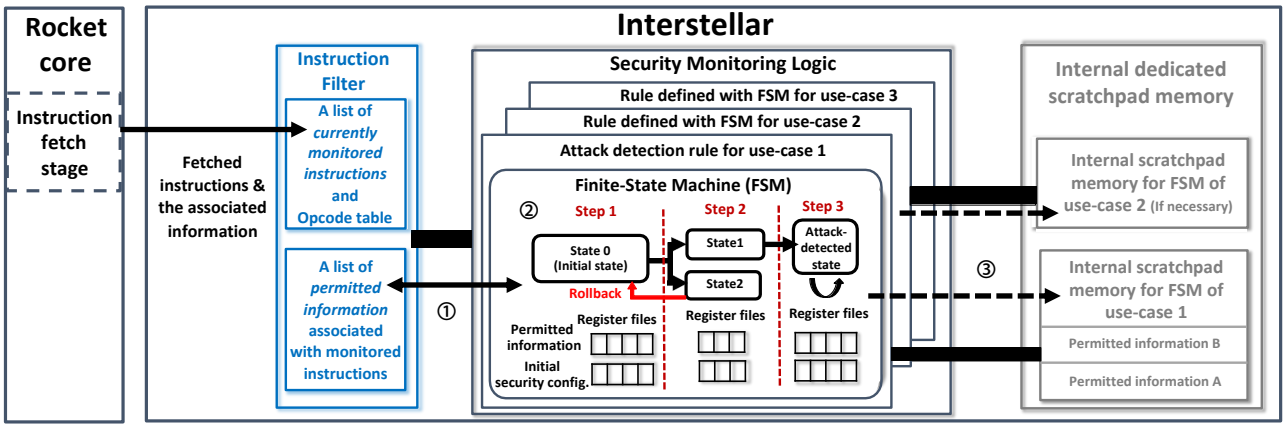

Figure 3: Detailed design of Interstellar. ①: The instruction filter passes only the required information associated with the monitored instruction to each FSM. ②: FSM evaluates the passed information and transits to the next state. ③: For long-term and repeated attack detection rules, permitted information can be stored in a dedicated internal scratchpad memory.

required information in the register files of each state that is associated with the monitored instruction. If the required information cannot be determined in the main core's fetch stage, each state of the FSM brings the necessary information (e.g., physical memory address) from the main core's stages where the information is determined (e.g., the memory stage) through the instruction filter. Then, each state utilizes the required information associated with the monitored instruction and the initial security configuration to detect malicious actions, and the state transition of FSM is made based on the detection result. Notably, the FSM processes internal register updates and state transitions within a single CPU clock cycle to achieve simultaneous detection. Also, if the state transition occurs, the FSM requests an update to the filtering list of the instruction filter, which is associated with the subsequent state.

Specifically, the state transitions within FSM work in the following order to detect malicious actions. Initially, to know which transition condition is met, the FSM evaluates the required information associated with the monitored instruction and the trusted initial security configuration generated during the boot. Depending on the evaluation result, the FSM can move to the next state, or it can stay in the current state for the next identical evaluation. In the case of staying in the current state, FSM stores the previous required information of the state into the FSM's internal scratchpad memory and then stores the new required information in the state's registers. In addition, when the monitored instruction is squashed on the main core due to branch misprediction or interrupts, FSM rolls back to the state in which the associated monitored instruction is squashed. The state rollback is accomplished by restoring the value of the previous registers that must be maintained for correct monitoring within a single CPU clock cycle. Lastly, if FSM reaches the attack-detected state through repeated state transitions and state restorations, Interstellar responds to the main core in a different way based on each attack detection rule.

Responding to the attack detection. Interstellar can take proper actions to the main core based on the attack detection result before the malicious instructions are committed on the main core. This is because a state of FSM finishes evaluating the detection result simultaneously, while the malicious instructions, which are utilized to perform attacks, are still on the main core. When malicious action conditions are met in the attack-detected state, Interstellar typically takes one of two types of actions: (1) invalidating the monitored instruction that performs the malicious action, or (2) halting the CPU pipeline.

The first action is taken if Interstellar can neutralize the attack only by invalidating the malicious instruction, and subsequent system execution on the main core can run without problems. For example, invalid memory access is performed with a single load or store instruction that accesses an unauthorized physical address, so that Interstellar can simply neutralize the attack by invalidating the problematic load or store instruction. The second action is taken if the subsequent system execution on the main core cannot be run normally after Interstellar invalidates the malicious instruction. In this case, the systems must be rebooted by the users. For example, if Interstellar invalidates a branch instruction against ROP attacks, then the system on the main core cannot be executed in normal flow.

4.4 Security Analysis for Possible Attacks

Attackers assumed in our threat model may bypass or disturb Interstellar's monitoring by learning the behavior of the attack detection rule defined with FSM. The attackers may learn the monitoring behavior of FSM by (1) directly accessing or (2) indirectly inferring the instruction filtering lists, the required information, and the initial security configuration in Interstellar.

First, privileged attackers cannot directly access the required information that is only accessible by FSMs at the microarchitecture level to monitor attacks. The reason is that, unlike the prior works [30, 33], Interstellar does not provide any privileged software interfaces, such as ISAs and interface functions, for privileged users to control the critical information of Interstellar. In addition, since the hardware components of Interstellar are also partitioned from the existing hardware components of the CPU, the attackers cannot access Interstellar's critical information utilized for detecting attacks with the existing ISAs and interface functions.

More importantly, the indirect attacker may infer how FSM monitor the attack by observing changes in the main core's microarchitecture to bypass Interstellar's monitoring, but the attackers are

infeasible to infer critical information, such as monitored instructions, to the best of our knowledge. First, the attacker cannot infer the attack detection rule by intentionally sending instructions to Interstellar and snooping their filtering results because Interstellar does not change any main core's microarchitecture according to the instruction filtering results at all. Secondly, attackers cannot infer whether Interstellar evaluates the fetched instruction to detect attacks by measuring the latency of the instruction until it is committed on the main core, similar to existing timing channel attacks. This is because Interstellar simultaneously completes evaluating the monitored instruction before the instruction is committed on the main core, and because of this, the monitoring behavior of Interstellar does not delay the commit of instruction on the main core, as shown in Figure 5.

5 Implementation and Use Cases

In this section, we explain how to implement Interstellar's FSMs, which define attack detection rules, considering the pipelined in-order Rocket core. Then, we present three use cases, which Interstellar detect and block various attacks on higher-privileged software, such as OS or TEE, by referring to existing detection methods [22, 37, 46, 50] or proposing concise detection methods achievable by Interstellar.

5.1 FSM Implementation

Attack detection rule. The attack detection rule is defined in the form of FSM with a sequence of states. The states have register sets that store the required microarchitectural information and the initial security configuration. In addition, the states have computational logic circuits that evaluate the state transition condition with the information in the registers. Importantly, the state's computational logic circuits must be implemented so that the state transition condition can be evaluated within a single CPU clock cycle, so as not to disrupt the Rocket core's pipeline. The reason is that Interstellar receives the fetched instruction and required information every cycle from the Rocket core's pipeline, and Interstellar is designed to be tightly coupled with the Rocket core to block attacks in a timely manner. In addition, each FSM has an initial state and begins the attack detection from the initial state after configuring the initial security information during the system's booting. The initial setup process for FSM can be implemented by setting a constant value in the state for the invariant configuration or obtaining the configuration value by monitoring specific instructions during the booting process for the variant configuration, as explained in the initial setup paragraph of subsection 4.3.

Instruction squashing handler. The attack detection rule of Interstellar receives the information associated with the monitored instructions from the Rocket core's pipeline. However, some fetched instructions monitored by FSM are squashed before committing on the Rocket core because of branch misprediction or interrupts. As a result, some previous registers' information utilized for detecting attacks must be preserved before the monitored instruction is committed to the main core. To handle instruction squashing issues, we implement an instruction squashing handler with FSM. The instruction squashing handler functions as an archive. The handler preserves the previous registers' values of the modified state upon receiving new information associated with the monitored instruction. It also retrieves information related to instruction squashing, such as interrupt flags, from the Rocket core to know whether the squashing occurs. If instruction squashing occurs, the instruction squashing handler restores the preserved FSM's state and internal register values. Conversely, if the instruction is committed successfully, the handler discards the preserved information. Meanwhile, Interstellar has a time window for handling instruction squashing issues. The shortest time interval between instruction squashing and receiving new information is a single CPU clock cycle. In this situation, if the time taken for handling the instruction squashing exceeds one CPU clock cycle, it is difficult to perform instruction monitoring without stalling the Rocket core pipeline. Fortunately, once instruction squashing occurs, it invalidates all instructions after the instruction that causes the squashing. This implies that squashing handling can be done by rolling back to the state that monitors the squashed instruction, without performing complex inverse operations on the state and registers. Since state and register rollbacks are accomplished by simply copying values from archived registers, instruction squashing handling is feasible in a single CPU clock cycle.

5.2 Use Cases

Monitoring unauthorized memory accesses. Prior unauthorized memory access attacks [21, 46, 47] bypass the memory access control for TEE by exploiting various vulnerabilities to leak data of the TEE from lower-privileged software. To demonstrate Interstellar's use case, we focus on detecting and blocking such unauthorized access attack scenario [47] that bypasses the memory access control of the TEE on the Rocket chip among the prior attacks. The reason is that other attacks are performed by exploiting particular vulnerabilities in the vendor's CPU, whereas the PMP attack is performed on the Rocket chip, where Interstellar is implemented. The attacks on the Rocket chip's TEE intervene in updating the PMP unit, which provides a memory protection mechanism for TEE, in order to access the memory region of the TEE from the attacker's application. The attacks make the systems skip CSR-related instructions, which are required to update PMP entries based on the initial security configuration, by injecting faults during runtime.

Interstellar can simultaneously detect and block such PMP poisoning attack for unauthorized access to TEE on the Rocket chip as follows. To get the initial memory protection information for TEE, Interstellar retrieves the physical address boundaries and permissions from the main core's PMP registers by monitoring CSRRW instruction (Atomic read/write CSR), which is used to set the PMP registers, during the system's booting. The information extracted from PMP is stored in the initial security configuration registers of FSM inside of Interstellar during the booting process. Since the isolated physical memory boundaries for TEE management are not modified after the booting period like security monitor of Keystone [39] and Intel SGX's PRM [26], Interstellar only trusts the initial PMP entries stored in the FSM's initial security configuration registers to detect attacks on TEE. After the initial configuration of Interstellar, Interstellar monitors the load/store type instructions. If a load/store instruction is received, Interstellar retrieves the target physical address from the main core's memory stage and evaluates it with the PMP information stored in the Interstellar. Notably, since the evaluation with multiple PMP entries can progress within a single CPU clock cycle using multiple comparison logic of the

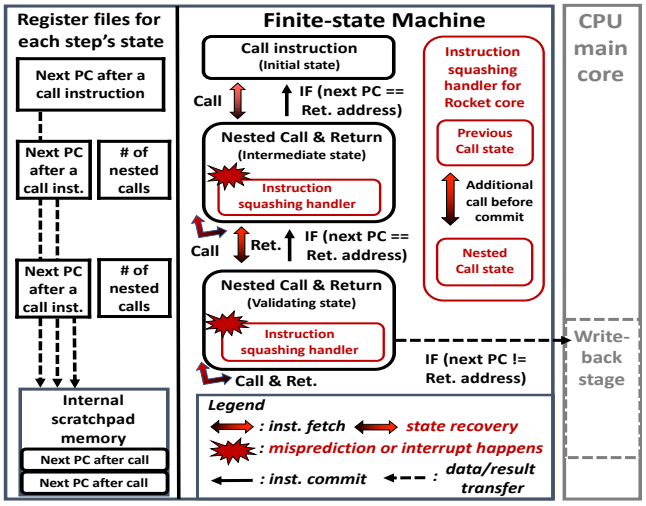

Figure 4: The FSM of Interstellar defining the shadow stack

FSM, the result of the evaluation can be responded to the main core before the load/store instruction is committed without stalling the main core's pipeline. If the evaluation results in unauthorized access, Interstellar will send an invalid memory access flag to the main core. The invalid memory access flag immediately squashes the unauthorized physical address's data loaded in the L1 data cache and invalidates the causing load/store instruction before it is committed on the main core. The cache invalidation is mandatory for preventing further unauthorized data leakage via cache side-channel attacks.

Monitoring ROP attacks. ROP attacks are representative attacks [18, 23, 40] that exploit memory safety vulnerabilities in the software to leak secret data from privileged software. The ROP attacks hijack the control flow of the privileged software by manipulating the return address in the function call stack for the privileged software. Among various defense mechanisms [22, 25, 49] against the ROP attacks, we implement the shadow stack [22] in the FSM of the Interstellar to detect and block the ROP attacks on TEE. The shadow stack ensures that the program does not escape the original control flow. The shadow stack records the return address in a secure shadow stack when the call instruction is executed and then compares the recorded address with the actual return address when the return instruction is executed. In particular, unlike PHMon's shadow stack [30] that must trust the OS kernel to detect the ROP attacks, Interstellar's shadow stack can be utilized to safely detect the ROP attacks on TEE from malicious OS kernel, such as Dark-ROP [40].

Interstellar's shadow stack mechanism is implemented with the FSM and internal scratchpad memory, as shown in Figure 4. To detect ROP attacks, FSM starts with the initial Call state, which monitors a call instruction among the instructions fetched from the main core. When the call instruction is fetched, the expected return address (e.g., the next *Program Counter (PC)* of call inst.) passed through the instruction filter is loaded in Interstellar's internal register. Then, FSM transitions to the Call & Return state by updating the filtering list to monitor the return and call instructions. Also, Interstellar stores the expected return address loaded in the previous state's register to the internal scratchpad memory after the call instruction is committed, and Interstellar sets the stack pointer to retrieve the expected return address later. When a call instruction is fetched in an intermediate Call & Return state, the expected return address is loaded in the internal register as before without state transition. If a return instruction is fetched, FSM transits to the Validating Call & Return state. In the Validating state, both call and return instructions are passed through the instruction filter in the same way as in the intermediate state, but Interstellar validates whether the control flow is hijacked or not at the same time. To detect the ROP attack, Interstellar pops the expected return address from the shadow stack of the scratchpad memory and compares the expected return address with the actual return address that can be manipulated by the ROP attacker. If the expected return address and the actual return address are the same, the state transits back to the intermediate Call & Return state. If they are different, Interstellar sends the ROP attack flag to the main core to halt the execution of the ROP attack.

Notably, the instructions fetched on the main core can be squashed for various reasons, so the execution flow information sent to Interstellar and the actual execution flow of the main core may not match unless the instruction squashing is handled. To correctly detect the ROP attack, the instruction squashing handler must be applied to the state that is repeatedly utilized to monitor the same behaviors. The reason is that the expected return address loaded in the state's register can be overwritten by the new expected return address of the next Call instruction before the instruction squashing is determined in the main core. Moreover, the instruction squashing handler is also mandatory for pushing the expected return address into the internal scratchpad memory without being overwritten by the next expected return address when the instruction is successfully committed. As shown in Figure 4, the instruction squashing handler of Interstellar is enough with two additional states to back up the expected return address for the correct attack detection, considering the structure of Rocket core's pipeline.

Monitoring microarchitecture timing side-channel attacks. So far, some works have proposed remote microarchitectural side-channel attacks [14, 35, 38], which exploit vulnerabilities in microarchitectural components on the RISC-V CPU (e.g., a shared cache or a branch predictor) and in target privileged software (e.g., OS kernel). We focus on the fact that most remote microarchitectural side-channel attacks must repeatedly measure the timing value related to the single behavior of the shared microarchitectural hardware in a short duration. This is because the measured time value has the noise from the uncontrollable hardware behaviors with the attacker's software. On the other hand, benign real-world applications and system software do not measure the precise timing values with such frequency. Especially, according to recent work to detect cache timing channel attacks [37, 50], they detect various cache timing attack methods with near 0% false positive rate and very high accuracy only using the threshold value of the frequency of time measurement that they discovered empirically.

To detect cache side-channel attacks on the RISC-V processor based on the core features of remote microarchitectural side-channel attacks, Interstellar monitors the RDCYCLE instruction of RISC-V (acting as the RDTSC instruction of x86), which the attacks utilized to measure the execution time of the microarchitectural hardware's behavior in a clock cycle unit with high accuracy. Also,

we define in FSM that the cache timing channel attack is detected if the time measurement period utilizing the RDCYCLE instruction is within threshold cycles [37]. The FSM for detecting cache side-channel attacks transits to the next state when the first RDCYCLE instruction is sent from the instruction filter. At the same time, FSM counts the clock cycle with the clock signal input synchronized with the CPU clock. If the next RDCYCLE instruction is sent from the instruction filter within the threshold cycle, FSM increases the violation counter. If the violation counter increases above the threshold, Interstellar determines that the attack is detected. Then, Interstellar sends a kill signal to invalidate all subsequently fetched RDCYCLE instructions before the instruction is committed on the main core.

6 Evaluation

6.1 Experimental Setup

We implemented Interstellar on a Rocket Chip [15] using the Chisel 3 [17]. Table 3 shows the experimental parameters for Interstellar and Rocket chip. We used Firesim [8] on the AWS EC2 F1 Xilinx UltraScale+ VU9P and AMD Vertex 7 FPGA VC707 Evaluation Kit [6] to evaluate the performance overhead of Interstellar-enabled Rocket chip. Firesim uses the Linux kernel v6.2.0 and the VC707 board uses the Linux kernel v4.15.0 equipped with the TEE functionality of Keystone. LMbench [9], twelve applications from MiBench [10], and seven applications from SPECint2006 [11] are utilized to evaluate performance overhead. In addition, we used the values provided by Xilinx Vivado [7] to measure the hardware cost: power and area. Lastly, Interstellar is synced with Rocket core's clock speed and includes all the FSMs for three use cases.

6.2 Functionality Validation

In this subsection, we validate the functionalities of Interstellar for each use case, described in subsection 5.2.

Physical Memory Protection. We reproduced the attack scenario where an attacker manipulates PMP entries and leaks secrets from an unauthorized physical address, similar to the prior unauthorized memory access attack [47]. To reproduce this scenario, we modified the Rocket core's PMP value that defines the TEE's physical address boundary, incurring the inconsistency between Interstellar's PMP information and Rocket core's PMP. This inconsistency problem also occurs in the situation where the prior attacker injects a fault in the PMP setting of the Rocket core by exploiting its target vulnerability. Then, we used the `devmem` function of Busybox [12] with OS privilege to access the unauthorized memory address, where the PMP originally protects. In the baseline Rocket core case, the unauthorized memory access attack on TEE [39] was successfully performed, although the target memory is under the PMP protection. On the other hand, in the Interstellar-enabled case, when unauthorized access to the originally PMP-protected memory occurred, Interstellar neutralized the unauthorized access attack.

ROP attacks in TEE. We used the attack scenario explained by DarkROP [40] to validate the shadow stack of Interstellar. We used Keystone [39] for TEE to reproduce the attack scenario against TEE on the RISC-V CPU. There is a function in the TEE part of the payload, which has `strcpy()` function causing a buffer overflow. The enclave code of the application lets an attacker manipulate the function's return address and hijack the control flow. For the Rocket core without Interstellar, the attacker successfully manipulated the return address of the vulnerable function and executed his target payload. However, in the Interstellar-enabled Rocket core case, the shadow stack of Interstellar detected the attack and prevented the application from jumping to the forged return address.

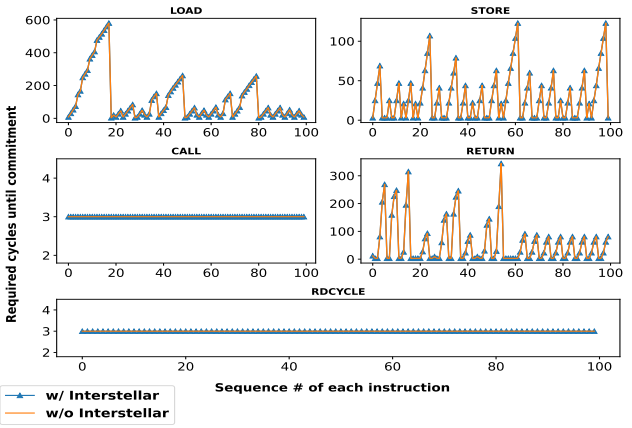

Figure 5: Required Rocket chip's clock cycles for each instruction until its commitment. Data was collected by running a RISC-V proxy kernel (LOAD, STORE, CALL, RETURN) or executing a simple application (RDCYCLE) on the Rocket chip emulator with and without Interstellar.

Pipeline	5-stage, in-order
L1 I-cache, D-cache	16KB, 4-way set-associative
L2 cache	512KB, 8-way set-associative, single bank
Interstellar's Block RAM	36KB, 576 64-bit-sized entries
Frequency (MHz)	50, 100 on VC707 board, 100 on Firesim

Table 3: Parameters for Interstellar and Rocket chip

Timing side-channel attacks. To validate the functionality of the timing attack monitor, we reproduced the *Flush+Reload* attack on the RISC-V CPU [35], which is a timing side-channel attack on the first-level cache. Interstellar increments the violation counter if the time interval between RDCYCLE instructions is within 100 cycles referring to the prior timing side-channel attacks on the first-level cache [35]. Also, if the violation counter exceeds 300, Interstellar considers the frequent execution of the RDCYCLE instruction as a timing side-channel attack. The attacker successfully cached and read the target information by measuring the memory access time when Interstellar is not employed. However, when the attacker tried to get the value by measuring the memory access time on the Interstellar-enabled CPU, the timing attack monitor of Interstellar identified the repeated memory access time measuring pattern and neutralized the following timing attack.

6.3 Performance Evaluation

We conducted performance evaluations to check the overhead induced by Interstellar. Then, we compare the performance of Interstellar with the performance of related works, PHMon [30] and ISA-grid [33]. Figure 6 shows the normalized execution time of SPECint2006, MiBench, and LMbench to the baseline Rocket core.

In the SPECint2006 and MiBench cases, even PHMon shows low overhead in the benchmarks (up to 3.4% and 5.1%), Interstellar shows much lower overheads in the benchmark (up to 0.68%

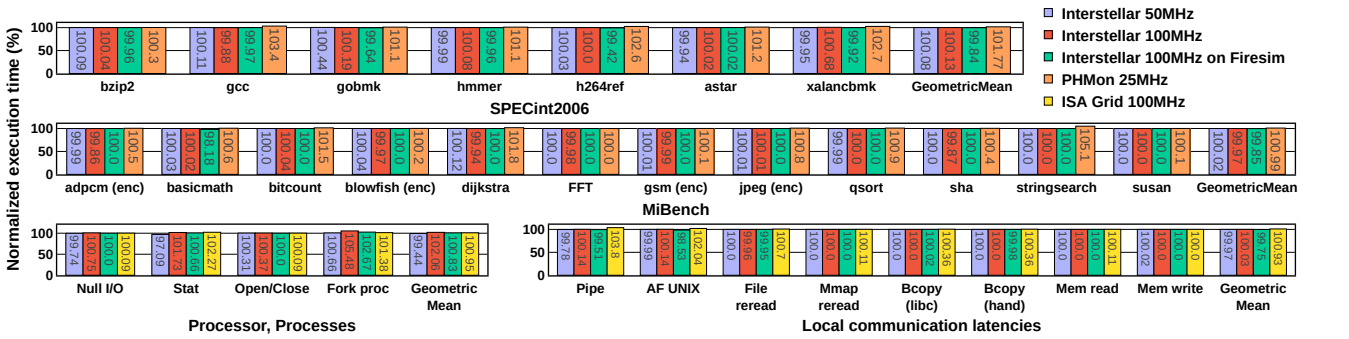

Figure 6: The normalized execution time of SPECint2006, MiBench, and LMbench. *Processor, Processes* and *Local communication latencies* are part of the LMbench results. Each execution time is normalized with the execution time on the baseline Rocket core, and 100% means the execution time on the Interstellar-enabled CPU is the same as that on the baseline CPU.

and 0.12%). Interstellar can show such extremely low overhead because Interstellar does not require any pipeline stall during the monitoring and attack detection process. This is possible because Interstellar can optimize each attack detection rule at the hardware logic level and determine whether the monitored instruction meets the attack condition before the instruction is committed on Rocket core. Figure 5 presents that Interstellar does not stall the execution of Rocket core's pipeline. Moreover, the results demonstrate that Interstellar is robust to plausible instruction timing channel attacks, discussed in subsection 4.4. Furthermore, the monitoring speed of Interstellar is not slowed down due to the main memory access, because Interstellar utilizes internal scratchpad memory, whose access latency is one CPU clock cycle, instead of the main memory. Thus, Interstellar shows better performance than PHMon in the SPECint2006 and Mibench.

In the LMbench case, most workloads show that Interstellar induces negligible overheads, up to 2.67%. Specifically, Interstellar shows much lower overheads of up to 0.13% when running on the experimental environment same as the ISA-Grid (VC707 board, 100MHz). Both the Interstellar and the ISA-Grid show such negligible overheads because both minimize the use of main memory and receive the monitored instructions at the Rocket core's frontend. However, the performance of Interstellar is slightly better in the same experimental environment. The reason is that ISA-Grid still requires main memory accesses to update the domain privilege cache, although ISA-Grid employs a domain privilege cache to reduce the main memory access.

6.4 Hardware Cost Analysis

We analyzed the hardware cost of Interstellar to explore how much additional power consumption and area overhead is induced. The power and area overheads of Interstellar presented in Figure 7 are obtained with all the FSMs for three use cases implemented in parallel. Figure 7 also shows the area overhead and power consumption of the Interstellar's each use case. The values are given by comparing the values of the Interstellar-enabled core and those of baseline Rocket core on the VC707 board.

Power. The additional power consumption induced by Interstellar is up to 34.10% compared to the baseline Rocket core. The total power overhead of Interstellar is almost the same as the sum of the power overhead of each use case because the Interstellar's FSMs for

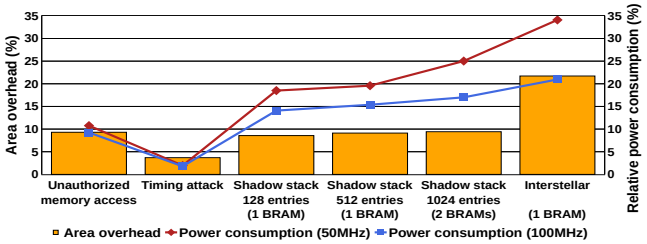

Figure 7: The relative hardware cost of Interstellar to Rocket core on VC707 board. Interstellar includes unauthorized memory access, timing attack, and 128-entries shadow stack.

use cases are executed in parallel. Also, the relative total power consumption of Interstellar decreases as the Rocket chip's clock speed increases (34.10% on the 50MHz core and 20.95% on the 100MHz core). This is because the increase in the power consumption of the baseline Rocket core is higher than that of the Interstellar as the Rocket chip's clock speed increases. The timing attack monitor, which has the simplest implementation among monitors, has the lowest power consumption, while the shadow stack has the highest power consumption because *Block RAM (BRAM)* is additionally used as the scratchpad memory for the shadow stack.

We also measured the shadow stack's power consumption for different numbers of entries to learn the effect of the shadow stack size. As the number of entries increases, the power consumption increases to 25.02% when the shadow stack has 1024 entries. As a BRAM can handle up to 576 shadow stack entries, two BRAMs are used when the shadow stack has 1024 entries. Thus, the increase in power overhead between the cases of 512 entries and 1024 entries is larger than the increase between the case of 128 entries and 512 entries because the number of BRAMs is changed.

Area. Area overhead represents the relative number of the *Look Up Tables (LUTs)* used by Interstellar compared to Rocket core on VC707 board, which are utilized for implementing registers and logic in Interstellar. The area overhead of Interstellar, where all three use cases are implemented together, is 21.72%. To analyze the area overheads associated with each use case, we also implemented each use case's FSM separately to measure the individual area overhead. As a result, for each Interstellar's use case, the timing attack monitor shows the lowest area overhead (3.69%) due to its simplicity. In addition, the memory access monitor has a slightly higher overhead (9.30%) than the default shadow stack (8.56%). This

is because Interstellar processes several physical address boundaries in parallel to monitor memory accesses in a single CPU clock cycle. The number of registers and logic LUTs used in the physical address-checking process increases proportionally to the number of physical address boundaries. Currently, Interstellar uses eight physical address boundaries, increasing the register and logic LUTs by a factor of 8.

Meanwhile, the Interstellar's shadow stack implements its internal scratchpad memory using BRAM, resulting in faster access speeds. Increasing the number of shadow stack entries can also increase the number of BRAMs based on the default BRAM size. Interestingly, Figure 7 shows that the number of LUTs does not increase proportionately to the number of stack entries. This is because neither the FSM nor the instruction filtering rule changes as the stack entry increases. Hence, the memory access monitor shows an area overhead higher than that of the shadow stack.

The prototype of Interstellar demonstrates an area overhead ranging from 3.69% to 9.30%, compared to 11% for PHMon [30] detecting a single attack. When detecting three attacks, the overhead for Interstellar increases to 21.72%, whereas 16% for PHMon with three matching units (MU). These hardware cost results for Interstellar are derived from the fact that Interstellar operates all attack detection rules in parallel and each rule utilizing its own registers and logic units, resulting in varying hardware costs for each attack detection rule, as illustrated in Figure 7. In contrast, PHMon incurs a consistent hardware cost for each matching unit, as its hardware design remains consistent regardless of the programmed use cases.

Although implementing defenses for additional use cases inevitably incurs additional hardware costs, Interstellar could meaningfully reduce overall complexity and costs by consolidating similar use cases, thereby enhancing scalability and efficiency. Currently, FSMs are separately designed to address distinct use cases. However, when multiple monitoring rules share comparable hardware logic units and operational contexts, the design of each FSM within Interstellar could be effectively merged into a single FSM with a modest quantity of additional hardware costs.

7 Discussion and Related works

In this section, we address some of the undiscussed aspects of Interstellar and compare Interstellar with additional related works.

7.1 Generality of Interstellar in terms of Other Microarchitecture's Features

Out-of-order execution. The FSMs of Interstellar were implemented on the Rocket core's in-order pipeline, whereas many commercial CPU cores have been designed with *Out-of-Order (OoO)* execution pipelines. Interstellar is subjected to some modifications for different microarchitectures of the core's pipeline, but we anticipate that Interstellar can support the OoO cores while ensuring simultaneous detection. This is because OoO pipelines still fetch instructions in order, and the commits of instructions are also done with the fetched order through reorder buffers. (i.e., OoO only occurs in the pipeline's execution stage). To achieve simultaneous detection, the part of the FSM that references hardware signals between the execution and commit stage of the core requires modification to allow the necessary data to be retrieved out-of-order from the main core in a timely manner.

Additionally, because of the additional hardware that supports OoO execution, OoO cores may require more cycles between the fetch and commit stages compared to the in-order cores. Interstellar can leverage the additional cycles to realize more complicated attack detection rules by utilizing the FSMs without harming the core's performance while achieving simultaneous detection.

Compatability on other ISAs. Interstellar is designed for the Rocket chip based on the RISC-V ISA, but Interstellar can also be utilized for other ISAs, such as x86_64 and ARM ISA. To support other ISAs, we need to modify Interstellar's instruction filter by referring to the opcode table and the register types of other ISAs. In addition, Interstellar is compatible with CISC ISAs as well as RISC ISAs while supporting simultaneous detection, even though the decoding of CISC ISAs is relatively time-consuming compared to RISC ISAs. This is because the Interstellar's instruction filter can be designed to decode CISC's instructions in a simultaneous manner during the main core's decode stage.

Impacts of faster and advanced cores. All components of Interstellar, including the instruction filter and FSMs, are synchronized with the CPU core's clock signal. Since the identical CPU clock signal is shared by the CPU core's pipeline and Interstellar, the attack detection process of Interstellar operates at exactly the same speed as the core's pipeline. Given that Interstellar and the core's pipeline operate at the same speed, even if the core's processing speed increases due to a higher clock frequency, Interstellar will still be able to fetch instructions and identify attacks within the specified time window of three cycles. Consequently, simultaneous detection remains feasible without pipeline stalls, even on advanced cores with higher frequencies. Besides, the advanced cores with a higher clock speed tend to utilize more hardware units to spread out many complex tasks over multiple clock cycles, but Interstellar's FSMs are not necessarily as complex as the pipelines on faster cores. This is because Interstellar only needs to retrieve the information needed for security checks without maintaining the entire states of the complex pipelines.

7.2 Practicality of Interstellar

Area overhead of Interstellar. The prototype of Interstellar was implemented on a Rocket chip, and the area overhead was measured relative to the Rocket core's area. Instead of Rocket core, if Interstellar is designed for advanced CPU cores, Interstellar's area overhead would translate into a relatively smaller overhead. This is because while the hardware components of an advanced CPU core are much larger compared to the simple 5-stage in-order Rocket core, the hardware components of Interstellar's FSM do not necessarily increase as much as the complex hardware components for supporting the advanced CPU core's capabilities.

According to the recent work [32] comparing the areas of Rocket core with other advanced cores, the ten-stage BOOM [3], which supports OoO execution and uses the same RISC-V ISA, has a 4.1 times larger area than the Rocket core when measured through ASIC. This increase in area is due to the presence of more pipeline stages compared to the five-stage in-order Rocket core. Also, support on the OoO execution results in a more complex structure and requires significantly more hardware components. Besides, similar to ARM's big.LITTLE architecture [1], Interstellar can be applied selectively to a CPU's particular secure core group.

Practical challenges in implementing FSMs. Once Interstellar's FSMs are implemented as logic circuits, users only can utilize the attack detection rules that are predefined in FSMs. Nonetheless, to allow CPU owners minimal composability, we can consider introducing CPU-specific instructions to change the detection rules in BIOS setup between the attack detection rules that are implemented on each FSM. Besides, the FSM-based design, while effective, may introduce complexity in the implementation and debugging process, particularly for large-scale systems with numerous attack detection rules. Implementing and verifying FSMs for all possible attack vectors in a large-scale deployment could be labor-intensive and error-prone. To reduce such manual efforts and decrease the likelihood of errors, we can consider the automated tools for generating FSMs based on high-level attack descriptions.

7.3 Related Works and Potential Use Cases

Notary. Notary [16] is a separate hardware security module that operates independently of software control. Notary is designed to securely execute security-sensitive applications without data or execution information leakage in multi-user environments by utilizing a deterministic start. Similar to Interstellar, Notary implements separate hardware security modules and is able to defend against information leakage through cache side-channel attacks. However, Interstellar and Notary differ in their goals and methods. Interstellar assumes all software can potentially perform attacks, while Notary focuses on inter-agent attacks. In addition, Interstellar reads instruction information to detect and defend against attacks, whereas Notary uses strong isolation and deterministic starts. These differences stem from Interstellar's focus on active defense through instruction monitoring, compared to Notary's passive defense through memory isolation and secure storage provision.

AMD PSP. The AMD *Platform Security Processor (PSP)* [13] is an isolated security co-processor within the Systems-on-Chip operating independently from the main processor's core, and it supports security features, such as hardware-validated boot and crypto acceleration. Moreover, PSP monitors the main processor's events to check whether the main processor is operating within the defined limits. Although both PSP and Interstellar seem to be similar because they are security modules operating in a hardware-isolated manner, their methodology for enhancing the system's security is different. While PSP focuses on isolating and providing security functionalities and checking the main processor's hardware status, Interstellar focuses on monitoring and timely blocking malicious actions performed on nearby main cores on a per-instruction basis.

Benefits of Interstellar and potential use cases. The high-level benefit of Interstellar compared to other methods such as setting the L-bit in PMP register is a better balance between security and flexibility: 1) the instruction-tracing hardware monitor's realms that are completely inaccessible from potentially vulnerable software and 2) timely and appropriate responses to various attack detection cases. Although setting the L-bit may achieve protection by locking the PMP entries from modification, the L-bit cannot be withdrawn unless resetting the system. Furthermore, it is difficult to use the L-bit for common attack prevention cases, such as privilege-based memory access control, because setting the L-bit enforces that PMP policy is applied uniformly to all the software regardless of its privilege. However, Interstellar can safely protect PMP entries from tampering attacks by storing the entries in its internal registers while retaining PMP's privilege-based protection functionalities. In addition, Interstellar can properly respond to the unauthorized data access via the PMP tampering by invalidating the data cache that might load the unauthorized data, preventing potential information leakage that exploits cache side-channel attacks.

Regarding a potential use case related to commercial CPUs, Interstellar could be utilized to detect the *Model-specific Registers (MSRs)*-abusing attacks [46] in Table 1, which injects fault into SGX enclave by manipulating the MSR that is utilized to adjust voltage and frequency of Intel CPU to restore crypto keys stored in the enclave. Interstellar can trace WRMSR privileged instruction, which is used to write values in certain addresses of MSR. Also, Interstellar can retrieve values stored in the EAX:EDX registers after the RDMSR or WRMSR instruction is fetched. As suggested in countermeasures of the attacks, the CPU's safe voltage levels can be defined after the chip testing, and Interstellar can use the predefined voltage levels to detect whether the abnormal voltage-frequency pair is written to the 0x150 MSR. During the booting process, Interstellar initializes the predefined safe voltage range in the FSM's initial security configuration registers. Then, Interstellar detects the attacks by comparing the safe voltage values with the values in the EAX:EDX registers when WRMSR instruction is sent to Interstellar. If the attack is detected, Interstellar can squash the instruction before the commit by sending a signal to the main core.

8 Conclusion

We presented the design and implementation of Interstellar, fully partitioned security monitoring logic from both the CPU's main core and privileged software. In addition, by introducing the FSM-based design, Interstellar can block instructions that carry out attacks in a timely manner without stalling the main core. We evaluated a prototype of Interstellar, including three defense use cases against the attacks on the privileged software, on the two different FPGA boards equipped with TEE-enabled and default Linux, respectively. Our evaluation shows that Interstellar's design effectively protects the system against the three attacks from the lower-privileged software that leak data by exploiting vulnerabilities in the higher-privileged software or shared hardware.

Acknowledgements

This research was fully supported by CySecuLab and partially supported by the NRF of Korea (NRF-2020R1A2C2101134), CSP, KEP, and KAIST-NYU project. This research was also partially supported by IITP under MSIT of Korea (RS-2024-00425503 and RS-2024-00440780).

References

[1] 2013. big.LITTLE Technology: Making very high performance available in a mobile envelope without sacrificing energy efficiency. https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf. Published: 2013.
[2] 2020 Jan. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.
[3] 2020 Jun. RISCV-BOOMDocumentation. https://docs.boom-core.org/_/downloads/en/stable/pdf/. Accessed: 2023-08-11.
[4] 2022 Jul. Chipyard Documentation. https://chipyard.readthedocs.io/_/downloads/en/1.7.1/pdf/. Accessed: 2023-08-11.
[5] 2022 Oct. Learn the architecture - An introduction to AMBA AXI. https://developer.arm.com/documentation/102202/0300/Channel-transfers-and-transactions.

[6] 2023 Nov. AMD Virtex 7 FPGA VC707 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html/#information.
[7] 2023 Nov. AMD Vivado - Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.
[8] 2023 Nov. Fast and Effortless FPGA-accelerated Hardware Simulation with On-Prem and Cloud Flexibility. https://fires.im/. Accessed: 2023-11-27.
[9] 2023 Nov. LMbench - Tools for Performance Analysis. https://lmbench.sourceforge.net/. Accessed: 2023-11-27.
[10] 2023 Nov. MiBench Version 1.0. https://vhosts.eecs.umich.edu/mibench/.
[11] 2023 Nov. SPEC CPU 2006. https://www.spec.org/cpu2006/.
[12] 2024 Jan. Busybox. https://busybox.net/. Accessed: 2024-01-09.
[13] Advanced Micro Devices, Inc. 2016. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 30h-3Fh Processors.* https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/programmer-references/52740_16h_Models_30h-3Fh_BKDG.pdf Revision: 3.06.
[14] Mahya Morid Ahmadi, Faiq Khalid, and Muhammad Shafique. 2021. Side-channel attacks on RISC-V processors: Current progress, challenges, and opportunities. *arXiv preprint arXiv:2106.08877* (2021).
[15] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), 6–2.
[16] Anish Athalye, Adam Belay, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2019. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 97–113.
[17] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R Avižienis, J. Wawrzynek, and K. Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221.
[18] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 1213–1227.
[19] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*. 30–40.
[20] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies*.
[21] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 991–1008.
[22] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy*. 985–999.
[23] Nicholas Carlini and David Wagner. 2014. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (SEC' 14)*. 385–399.
[24] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. 2022. ReZone: Disarming TrustZone with TEE Privilege Reduction. In *31st USENIX Security Symposium, USENIX Security 2022*. USENIX Association, 2261–2279.
[25] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. 2014. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society.
[26] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016), 86. http://eprint.iacr.org/2016/086
[27] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium, USENIX Security 16*. USENIX Association, 857–874.
[28] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*. 20–38. https://doi.org/10.1109/SP.2019.00021
[29] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (SEC' 14)*. 401–416.
[30] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. 2020. PHMon: A programmable hardware monitor and its security use cases. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 807–824.
[31] Leila Delshadtehrani, Schuyler Eldridge, Sadullah Canakci, Manuel Egele, and Ajay Joshi. 2017. Nile: A programmable monitoring coprocessor. *IEEE Computer Architecture Letters* 17, 1 (2017), 92–95.
[32] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. 2021. A comparative survey of open-source application-class RISC-V processor implementations. In *Proceedings of the 18th ACM international conference on computing frontiers*. 12–20.
[33] Shulin Fan, Zhichao Hua, Yubin Xia, Haibo Chen, and Binyu Zang. 2023. ISA-Grid: Architecture of Fine-Grained Privilege Control for Instructions and Registers. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Association for Computing Machinery, Article 15, 15 pages.
[34] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 21)*. 275–294.
[35] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. 2023. A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs. In *44th IEEE Symposium on Security and Privacy*.
[36] Mohammad A. Islam, Shaolei Ren, and Adam Wierman. 2017. Exploiting a Thermal Side Channel for Power Attacks in Multi-Tenant Data Centers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, 1079–1094.
[37] Arsalan Javeed, Cemal Yilmaz, and Erkay Savas. 2021. Detector+: An approach for detecting, isolating, and preventing timing attacks. *Computers & Security* 110 (2021), 102454.
[38] Anh-Tien Le, Trong-Thuc Hoang, Ba-Anh Dao, Akira Tsukamoto, Kuniyasu Suzaki, and Cong-Kha Pham. 2023. A cross-process Spectre attack via cache on RISC-V processor with trusted execution environment. *Comput. Electr. Eng.* 105 (2023), 108546. https://doi.org/10.1016/J.COMPELECENG.2022.108546
[39] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 38:1–38:16.
[40] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 523–539.
[41] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 557–574.
[42] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium, USENIX Security 16*. USENIX Association, 549–564.
[43] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, 973–990.
[44] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015*. IEEE Computer Society, 605–622.
[45] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S Müller. 2010. Characterizing the energy consumption of data transfers and arithmetic operations on x86- 64 processors. In *International conference on green computing*. IEEE, 123–133.
[46] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1466–1482.
[47] Shoei Nashimoto, Daisuke Suzuki, Rei Ueno, and Naofumi Homma. 2020. Bypassing Isolated Execution on RISC-V with Fault Injection. *IACR Cryptol. ePrint Arch.* (2020), 1193. https://eprint.iacr.org/2020/1193
[48] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. TrustZone Explained: Architectural Features and Use Cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. 445–451. https://doi.org/10.1109/CIC.2016.065
[49] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. USENIX Association, 447–462.
[50] Musa Sadik Unal, Arsalan Javeed, Cemal Yilmaz, and Erkay Savas. 2022. Hyperdetector: Detecting, isolating, and mitigating timing attacks in virtualized environments. In *International Conference on Cryptology and Network Security*. Springer, 188–199.
[51] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.
[52] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. 2023. SHELTER: Extending Arm CCA with Isolation in User Space. In *32nd USENIX Security Symposium, USENIX Security 2023*. USENIX Association.