Technical Report

UCAM-CL-TR-980
ISSN 1476-2986

Number 980

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Protecting enclaves from side-channel attacks through physical isolation

Marno van der Maas

March 2023

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 2023 Marno van der Maas

The following report is licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence:

https://creativecommons.org/licenses/by/4.0/

UNIVERSITY OF CAMBRIDGE and the Coat of Arms are registered trade marks of The Chancellor, Masters, and Scholars of the University of Cambridge (“University Marks”). For the avoidance of doubt, the University Marks are not included in this Creative Commons licence.

This technical report is based on a dissertation submitted September 2021 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

The digital world is taking an increasingly crucial role in our lives. Digital systems control our calendars, how we gain access to our devices and even the vehicles we use for transportation. It is therefore no surprise that security solutions like trusted execution environments (TEEs) have been introduced in many systems ranging from small embedded networking devices to large server racks. One of the main challenges of this ever growing functionality is keeping the trusted computing base (TCB) small and manageable. Enclave systems are a way to do exactly that: they allow applications to run on the same system as a rich operating system (OS) while ensuring the confidentiality and integrity of enclave data.

In this thesis I explore the difficulty in protecting enclaves from side-channel attacks in the face of privileged software. I propose a threat model, a methodology to analyze side channels and a new enclave system that adheres to this threat model. Due to the complexities of modern superscalar processors, I conclude that it is undesirable to run enclaves on the same cores as untrusted software due to the performance degradation this would have on regular applications. My new enclave system uses a heterogeneous multi-core processor to physically isolate enclaves on secure cores while regular applications run on fast cores. I show that this system works with a conventional OS by implementing a Linux driver that facilitates management of enclaves and communication between untrusted applications and enclaves. The enclave subsystem only requires a small TCB: a trusted management shim to interface the Linux driver with the enclave hardware. I evaluate hardware implementation approaches in simulation and on a field-programmable gate array (FPGA). The evaluation shows that this system is reasonable in communication overhead, memory footprint, runtime and hardware area. Thus, physical isolation is a feasible way to protect enclaves from side-channel attacks in modern enclave systems.

Acknowledgments

Computer architecture and security are my passions and I am grateful that I have had the opportunity to do a PhD in this area. I want to thank everyone at the Computer Laboratory and the Computer Architecture Group for all the chats and support. A big thank you to Professor Simon Moore, whose supervision and feedback were fundamental in fulfilling this work. Thank you to Alexandre Joannou, Robert Kovacsics, Matthew Naylor, Zahra Tarkhani and Robert Watson for their insights and help. I specifically want to thank the following people for reading through my thesis and giving me feedback: Nathaniel Filardo, Franz Fuchs, Ivan Gomes Ribeiro, Emily Goodacre, Theo Markettos, Simon Moore, Peter Neumann, Peter Rugg and Jonathan Woodruff. Thanks also to my examiners, Robert Mullins and Daniel Page, for your time and feedback of my thesis.

Having the financial means to complete my research has been essential. I have been funded by a Qualcomm European doctoral fellowship and a doctoral studentship from the UK's engineering and physical sciences research council.

Every system that I developed was made possible by free and open source software. An incomplete list of open source projects that I used are: the Bluespec compiler, CHERI Toooba, FreeBSD, the GNU toolchain for RISC-V, Linux, RiscyOO and Spike. I want to take this opportunity to thank everyone who has contributed to the open source community.

Specifically to the friends, staff and members of my college Clare Hall; thank you for providing a home away from home, especially the Clare Hall Boat Club for helping me not leave my body behind as I go on this mostly mental endeavor. I also want to thank my wife, parents, siblings, family and friends who helped me stay grounded during this sometimes stressful process.

Sadly I cannot name everyone who has helped me get this far. So I want to thank everyone who has had an impact on my life during and before this PhD. Even if I do not name you specifically, I am a firm believer that every interaction has an impact and that the sum of all of those experiences has helped me get to this point.

Ik wil mijn familie en vrienden in Nederland en over de hele wereld bedanken voor alle steun. Aan mijn ouders, Eveline en Theo, jullie hebben er voor gezorgd dat ik het beste uit mezelf kon halen.

Contents

List of figures

List of tables

Glossary

API	application programming interface
AXI	advanced extensible interface
BRAM	block random-access memory
CAS	column address strobe
CHERI	capability hardware enhanced RISC instructions: Watson et al. write that CHERI “extends conventional processor [ISAs] with architectural capabilities to enable fine-grained memory protection and highly scalable software compartmentalization” [5].
CPU	central processing unit
CSR	control and status register
CTC	CHERI tag controller
DMA	direct memory access
DRAM	dynamic random-access memory
ELF	executable and linkable format
FMI	fast memory interface
FPGA	field-programmable gate array
GPU	graphical processing unit
IOMMU	input-output memory management unit
ISA	instruction set architecture

LLC	last-level cache
LUTs	look-up tables
OS	operating system
PUF	physically unclonable function
RAM	random-access memory
RISC-V	reduced instruction set computer five: Waterman et al. write that “RISC-V (pronounced ‘risk-five’) is a new ISA that was originally designed to support computer architecture research and education, but which [they] now hope will also become a standard free and open architecture for industry implementations” [6].
ROM	read-only memory
RSA	Rivest–Shamir–Adleman: An algorithm that uses the difficulty of the factoring problem for asymmetric cryptography.
RVWMO	RISC-V weak memory ordering
SoC	system on chip
TAP	trusted abstract platform
TCB	trusted computing base
TEE	trusted execution environment
TLB	translation look-aside buffer
TSO	total store order

Chapter 1

Introduction

Operating systems (OSs), hypervisors and other privileged code are becoming increasingly complex, and security researchers discover vulnerabilities in these codebases in higher numbers than a few years ago [7]. Many applications run on OSs that are untrustworthy and have their own requirements for trustworthiness. For this reason, trusted execution environments (TEEs)[1] provide a way to isolate application code from the underlying OS. There are many ways to provide this isolation, from providing a new privilege level like Arm TrustZone [9] to providing a completely separate security chip like Apple's T2 [10]. Another approach is to rewrite the whole operating system like with seL4 [11] or the antikernel [12]. In this thesis I focus on TEEs that allow applications to guarantee the integrity and confidentiality of their data, while still relying on a conventional OS or other privileged code to do resource management. TEEs like Intel SGX [13] have become more and more popular in the last few years; AMD [14] and Arm [15] have launched similar systems. I refer to applications that are protected in these type of TEEs as *enclaves*. This work is motivated by the desire to protect enclaves from side-channel attacks and that, in the age of dark silicon, it makes sense to create dedicated hardware accelerators for security use cases.

To make a successful enclave system, there are a few main obstacles. For example, modern superscalar out-of-order processors open up innumerable side channels. These side channels have become higher bandwidth and increasingly exploitable over the course of my PhD through the discovery of transient execution attacks like Spectre and Meltdown [16]. Additionally, enclave threat models include privileged code, which make side channels easier to exploit because of access to high-precision counters, direct cache control, etc.

[1] It is curious that the TEE acronym uses the word "trusted" instead of "trustworthy." Peter Neumann writes that "trustworthiness implies that something is worthy of being trusted. Trust merely implies that you trust something whether it is trustworthy or not, perhaps because you have no alternative, or because you are naïve, or perhaps because you do not even realize that trustworthiness is necessary, or because of some other reason" [8]. I use the word "trusted" throughout this thesis because that is the convention in the literature.

Protecting programs that are collocated on such cores is infeasible without significant performance loss. Hardware mitigations cause the removal of many microarchitectural performance features or the addition of expensive flushing instructions [17]. These observations become evident in real systems by the discovery of many side channels in Intel SGX and other enclave systems, some even exposing the platform keys that are the foundation of secure remote attestation [18].

This vulnerability to side-channel attacks occurs because the industry lacks a good understanding of enclave threat models. For example the threat model of AMD SEV initially only protected against a passive attacker [14]. This is unrealistic when privileged code is part of the attacker model. There is thus a need for a different threat model for enclave systems.

To address these problems, I make the following contributions in this thesis, some of which I published in a paper titled "Protecting Enclaves from Intra-Core Side-Channel Attacks through Physical Isolation" in 2020 [1]:

- A threat model for enclaves that includes software-based side-channel attacks (Chapter 2), which are usually completely or partially excluded by previous enclave systems.

- A methodology for converting side channels into direct channels, which models an upper bound on the possible information leakage via this channel (Chapter 3). This can be used to study side channels, to propose mitigations for them and generally enhance the structure of a process that is usually dependent on human intuition and manual checking.

- A system for physically isolating enclaves onto separate secure cores, called Praesidio[2] (Chapter 4). This combines the notion of physically isolated cores and allowing mutually distrusting code to run in the TEE while delegating resource management to an untrusted OS.

- A security analysis of this proposed system and what it guarantees to enclaves (Chapter 5). This includes a comparison to an abstract enclave platform, an analysis of the trusted computing base (TCB) and a discussion of important attacks.

- An evaluation of one possible embodiment of such a system using a Linux driver, a small dedicated TCB and a modified RISC-V instruction set simulator (Chapter 6). This shows that running enclaves on secure cores still allows user applications to launch, interact with and attest secure compartments despite running on top of an untrusted OS.

- An evaluation of the memory protection mechanisms in the Bluespec SystemVerilog hardware description language and using a superscalar out-of-order RISC-V processor

[2] Praesidio means "protection" in Latin. This name was suggested by my supervisor Simon Moore.

based on MIT's RiscyOO (Chapter 7). This is crucial to show that Praesidio can realistically be added to a system without significantly degrading the performance of the normal application cores.

More broadly my system proposal gives application developers a choice of trade-off between security and performance. As a heterogeneous multi-core system, it allows developers to make their own choice on the security to performance scale. This thesis aims to show how protecting enclaves from side-channel attacks is paramount for any practical enclave system and that physical isolation is an effective, practical and performant solution to this problem.

Chapter 2

Background and motivation

2.1 Introduction

The need for trusted execution environments (TEEs) and enclave systems is apparent from the increasing popularity of these systems in computing today. Several previous works have considered enclave systems, but given the divergence in protections provided by each of these systems it is hard to create a consolidated view of what an enclave developer can expect. I propose a threat model which includes side-channel attacks that can be launched by any software-based attacker. Throughout this chapter I discuss the motivation for creating enclave systems and the contributions of previous enclave systems.

2.2 Trusted execution environments

TEEs are appearing in use cases from embedded network devices to server rack machines. For example Sancus is a way to allow remote third-party software to be installed on networked devices [19]. It has no software in the trusted computing base (TCB), and it guarantees that each application is isolated from each other. In terms of server rack systems, AMD SEV [14] and Intel TDX [20] have shown that commercial companies are interested in providing hardware-guaranteed isolation for virtual machines. Additionally, Arm TrustZone [9], smart cards [21] and trusted platform modules [22] have shown the importance of TEEs in consumer electronics. It is no surprise that TEEs are gaining popularity since they provide a way to protect applications from the rest of the system.

2.3 Privileged attacks on applications

Ever since the advent of multi-user systems, computer engineers have pursued methods of isolating and protecting users from each other. In the case of Multics [23], this is done using privileged rings, where the more privileged ring is reserved for the kernel and is in charge of isolating users from each other. This model of protection is still prevalent today. However, the ever-growing complexity of modern kernels and computing systems means that this model of protection is becoming inadequate. Successful attacks on kernels such as the Linux kernel are becoming more and more frequent [24]. The kernel can still manage isolation of processes, but there now also needs to be a mechanism by which applications can protect themselves from the kernel.

2.4 Background on side channels

Side-channel attacks in computer architecture can use architecturally exposed features or use microarchitectural details to extract information from a victim. For example cache timing can expose encryption keys [25] and execution time can leak information about a process' control flow.

There is a distinction between software and physical side-channel attacks. Software-based side channels only require software access to a device – i.e. the attack can be done with software only. Physical side channels require physical access to the device. A classic example of a software-based side channel is a "prime and probe" attack using shared cache lines. The attacker can measure the access time of a cache miss versus a cache hit and use this to detect whether a cache eviction took place [25]. Power analysis is an example of a physical side channel. An observer of the power consumption of a chip can perceive the activity of a process depending on what type of work the processor is executing [26]. Other examples of physical measurements that can be side channels are voltage, current, electromagnetic fields, photon emissions, temperature, and sound. However, because requiring physical access to a device is less scalable than just requiring software access, enclave systems should primarily focus on protecting against software-based side channels.

Many software-based side channels transmit information through the contention of shared resources between processes. These resources are usually microarchitectural and contention creates a timing side channel that the attacker can use to extract information from the victim. Examples of shared resources are cache lines, execution units and branch predictors.

2.4.1 Inter-core side channels

Inter-core side-channel attacks are those that can be launched between processes running on separate cores. In this section I consider only side channels that can be exploited by privileged software and exclude any side channels that require physical access. These parameters align with the threat models of modern TEEs like Intel SGX [13, 17, 27–29].

Cache collisions

Cache collision attacks depend on loading a cache line and measuring whether the line is loaded back into the cache by the victim [25,27]. Cache evictions are necessary because caches are smaller than the backing memory that they are caching. Upon loading a new line, it might be necessary to evict a line that is currently in the cache. Whether an eviction is necessary depends on cache associativity and the current cache state. The cache's eviction policy affects which cache line gets evicted. Cache collisions happen because caches are typically not fully associative. In this case, each address is mapped into one particular cache line or set, which means multiple addresses map to the same cache line or set, causing a collision.

Consider the "prime and probe" attack where the attacker and victim run on separate cores and share the last-level cache (LLC). This attack relies on two primitive operations: evicting a victim's cache line (priming), and detecting whether the cache line has subsequently been loaded back in after running the victim (probing). To prime the cache, the attacker needs to find a set of addresses which collide with the victim address and cause an eviction when loaded. To probe, attackers typically use a timer to count how many cycles occur between two instructions. Cache misses require extra cycles before they return the load value, so a higher cycle count means that there was a miss and a lower count means there was a cache hit.

DRAM row buffer collisions

DRAM row buffer collisions use the fact that each dynamic random-access memory (DRAM) bank has a set of sense amplifiers which contain the previously read row. It takes less time to read from a row that already resides in the sense amplifiers than it does if the amplifiers need to be populated with a new row [30]. The attacker can thus detect whether a victim has re-opened a row in a bank, potentially leaking sensitive data.

One way to close this channel is by making the DRAM controller take constant time [17]. This means that even if a row is open in the sense amplifiers, this does not service the memory request faster than if the data resides in a closed row. This mitigation blocks the probing part of this attack, but increases DRAM access time to already opened rows. In one of Samsung's DDR4 DRAM chips this increases access time to already opened rows by between 40% and 50% [31].

Cache requests

The time it takes to serve a cache request depends on how busy the cache is. The attacker can gain data due to having access to the same buffer as the victim has, which leaks information about when the victim uses this buffer. A mitigation for this attack is creating separate buffers for each core and making a cache arbiter that handles requests from each channel in a round robin fashion [17]. This mitigates the attack by creating separate channels for the attacker and the victim as well as guaranteeing there is no timing dependency between these channels. The round robin arbiter has the downside that cache requests are delayed for cores that are not currently selected, even if there is no incoming request from the currently selected core.

Dram requests

The time to serve DRAM requests can vary just like cache requests. DRAM contention can cause cache request buffers to become full and leak information across domains [17]. Bourgeat et al. [17] sizes the cache request buffers so that DRAM can handle all outstanding requests at the same time. This means that the maximum number of outstanding cache requests is dependent on how many concurrent requests the DRAM controller can handle, which also needs to be designed to prevent timing side channels like DRAM row buffer contention (Section 2.4.1).

Dram bit leakage

The physical implementation of DRAM makes it possible for slight changes in charge to adjacent rows when performing a DRAM write, which can cause bit flips to occur [32]. If an attacker has access to adjacent rows, it can both tamper and eavesdrop on adjacent rows. Mitigations to this attack usually eliminate the ability for consecutive writes to affect the same row, like increasing refresh rate or remapping addresses [33]. Introducing integrity guarantees through cryptography stops the attack by allowing the victim to detect tampering. Merkle hash trees are one such cryptographic technique that is used to protect DRAM content in TEEs [13, 28, 34].

2.4.2 Intra-core side channels

Intra-core side channels require both the victim and attacker to be collocated on the same core.

Page access tracking

Page faults can be used to track which pages are being accessed by a victim [35]. When a memory translation is missing in the translation look-aside buffer (TLB) a page fault occurs. Because enclave attacker models include privileged code, exception handling tells the attacker when a TLB miss happens. Similar attacks are possible by monitoring the “access” and “dirty” bits in the page table. A mitigation for both of these attacks is having a trusted runtime handle page faults and page table management [27].

Execution unit usage

Execution units [36,37] and micro-op caches [38] are shared between threads in a core that allows simultaneous multithreading. Contention on these resources can leak information from the victim to the attacker. Mitigations to these attacks include disabling simultaneous multithreading [17,28] and having dedicated execution units per thread.

Branch shadowing

The branch predictor can be used to extract information about a victim’s control flow because victim’s branches affect the prediction of other code including that of a potential attacker [39–41]. Purging all branch predictor state between any two programs running [17] is one way of mitigating this attack and partitioning the branch predictor per process is another.

2.5 Transient execution

Side channels have become more important in the duration of my PhD due to the discovery of transient execution attacks. To avoid stalling and improve performance, modern processors speculatively execute instructions before they are certain of whether those instructions should be executed or not. In case this speculation was incorrect, the processor rolls back and reverses the effects of those instructions. Transient execution attacks make use of these misspeculations to infer information which they are unable to access through regular side channels [16,18]. Many microarchitectural performance features have been shown to leak information; these features include the cache, TLB, branch target buffer, branch history buffer, pattern history table, return stack buffer and store-to-load forwarding logic [16]. Transient execution attacks have also evolved from simply observing leakage from speculation to actively affecting what goes on in speculation, such as load value injection [42]. As attackers find more microarchitectural features to exploit and more creative ways to guide speculation, it becomes increasingly important to consider the security impact of complex microarchitectural features. This is especially true as speculative, out-of-order processors become more complex.

Transient execution attacks are also important due to a growing number of TEEs that protect applications from a privileged attacker (e.g. the operating system (OS) or hypervisor) [13, 17, 27–29]. These solutions implement increasingly costly features like partitioning caches [27] and expensive flushing instructions [17]. A privileged attacker can perform side-channel attacks more easily than user-level applications due to having control over scheduling and access to precise hardware timers. One extra class of attacks that is available to privileged code is controlled channel attacks, for example privileged code can evict TLB entries to determine memory access patterns [35]. For TEEs it is thus more important to consider transient execution attacks as part of the threat model.

2.6 Enclave use cases

Instead of focusing on isolating complete virtual machines, enclave systems focus on only protecting the sensitive part of an application. The secure compartment can then be attested remotely. Evtyushkin et al. say this "model is similar in principle to the design of Hardware Security Modules" [43], but without the need for a completely separate device. Another use case described by Evtyushkin et al. is periodically testing whether a machine or piece of software has been tampered with. There are a few concrete use cases for lightweight compartments, which I divide into the following categories: a remote third party wanting to check the integrity of the software running on a user's device; an application wishing to keep data secret like a key, software IP or private user data; a user wanting to use a third party service without disclosing their private data; and a public ledger system needing a proof of environment. The following subsections contain examples of use cases that fall into these categories.

2.6.1 Remote software integrity checking

Remote software integrity checking is similar to secure machine attestation [43]: One way to ensure privacy is to keep all private data on a local device and allow external services to run applications on this device that ensure that only limited data is revealed to these external service providers. This is the idea with Databox [44]. Databox gives users complete control over their data, but also makes it so that the service provider no longer controls the hardware on which their applications run. This introduces a new problem of how the service provider can ensure that the correct application is running on the Databox. Lightweight enclave environments can be used to ensure the software integrity of the applications. Databox is envisioned as a smart home device, which can be a separate box, integrated in a smart TV or a smart thermometer. The same principle could also be used for smart cars, smart phones or other internet of things devices.

Field-programmable gate array (FPGA) accelerated cloud computing has a similar problem to Databox, since clients want to generate their own encrypted bitstreams to protect their intellectual property, but the cloud provider must ensure that the netlist is checked for malicious patterns before the bitstream is generated. Zeitouni et al. propose running a virus scanner equivalent for FPGAs inside an enclave to solve this problem [45]. This way the client is assured that no intellectual property leaves their device unencrypted and the cloud provider is assured that the proper checks are done before allowing a bitstream on their machines.

2.6.2 Keeping application secrets

Applications sometimes need to embed secrets that should not be leaked. Examples include keys like private authentication keys, keys to access a service or digital rights management keys (e.g. Microsoft Playready [46] or Google Widevine [47]). Another use case is if an application has sensitive intellectual property embedded in it. In these cases, application providers want to keep the sensitive intellectual part in an enclave environment to prevent disclosure.

Additionally, user authentication can be done using enclaves. Existing solutions use are usually based on the FIDO specification [48], which can be implemented in a separate device or a software solution. Locating this authenticator in an enclave avoids having to carry around a physically separate device while still protecting the data from compromised software.

2.6.3 Private cloud computation

One example is a private membership test. This means that a user wants to check whether a value is in the database on the server without disclosing the value that is being checked. One specific use case for this is antivirus programs. There are privacy concerns with disclosing which programs are installed on a person's computer, since this can easily identify single individuals. Antivirus providers usually keep a database of hashes of known malicious programs. So checking whether a hash of a currently installed program is included in the malicious database of an antivirus provider without disclosing the hash itself is an example of where a private membership test can be used [49].

2.6.4 Public ledger with proof of environment

The idea for public ledgers is to be able to ensure transparency. Making public ledgers distributed brings a number of extra problems with it, for example resolving race conditions and establishing trust. Race conditions are usually resolved by the principle that the longest ledger wins. To underpin trust current systems use proof of work and the value of a cryptocurrency [50]. Enclaves allow a different type of trust by being able to prove that work has been executed in a trusted environment. We can use the attestation in enclaves to prove to the outside world that an update to a public ledger has been done by trusted enclave code. A use case in which this would be useful is in a business setting where multiple different parties are involved in a supply chain. One example of this is a food supply chain, where there are multiple farms, processing plants, transportation companies and distribution points [51]. Instead of having to create a secure network between all of the companies in this industry, we can allow this ledger to be shared freely among machines in different security domains and we only require a trusted enclave to run on the machines that update the public ledger.

2.7 Existing enclave systems

Enclave systems offer a way to protect applications from the kernel. Intel SGX is one of the most well known enclave systems and one that is widely available. However, the idea of TEEs that cannot be influenced by the kernel has been explored before, and previous research provides various solutions to address subsets of this problem.

2.7.1 Memory tagging

Timber-V [29] introduces a "new tagged memory architecture featuring flexible and efficient isolation of code and data on small embedded systems." Their main contribution is implementing a memory protection unit that uses small and fine-grained memory tagging of two bits per 64 bits of data. They use this tagging scheme to implement stack and heap interleaving.

In similar contexts Arm TrustZone and CHERI capabilities have used memory tagging for various security purposes. Arm TrustZone [9] marks secure world ownership of memory including cache lines with a single bit. CHERI [52] uses tagging to indicate whether a word is a valid pointer or not.

2.7.2 Encryption

XOMOS [53] runs on the XOM (execute only memory) processor. It uses hardware tagged compartment identifiers for each program and encrypts the data for each compartment with a separate key. It also introduces the idea of encrypting cache lines as they are evicted from the LLC to working memory a.k.a. DRAM. Aegis [34, 54] describes different protection mechanisms for memory, by splitting memory into dynamic and static regions as well as insecure, integrity-protected and confidentiality-protected regions. It improves the dynamic integrity protection over XOM by being robust against memory replay attacks, which it does by introducing the idea of using Merkle hash trees for integrity checking. OASIS [55] is an instruction set extension that uses the concept of Cache-as-RAM to ensure an isolated execution environment, where any data overflowing the cache needs to be encrypted before going to DRAM. One of the weaknesses of OASIS is that it has a significant overhead for programs that have a memory footprint which exceeds the cache size. Iso-X is the first solution to have no cryptography between cache and DRAM but to have cryptography between DRAM and disk instead [43].

2.7.3 Physically unclonable function

Aegis uses physically unclonable functions (PUFs) to embed a secret in the chip with low overhead. PUFs use variation in semiconductor manufacturing to create circuits that generate a unique value for each chip. These unique values can be used as a seed for unique cryptographic keys that are used in many enclave solutions. Like Aegis, OASIS uses PUFs to embed a secret, but it makes trusted non-volatile memory optional and, due to the addition of an owner seed in the key generation process, makes a trusted manufacturing environment optional.

2.7.4 Shadow page tables

Bastion [56] provides protection on a per page basis by using shadow page tables and nested paging. Whenever a change is made to the page table of the processor, a trusted routine checks whether this change is not assigned to an enclave which are maintained in the shadow page table. This is the main alternative to memory tagging and is used by Intel SGX [13] and Sanctum [27] as well.

2.7.5 Attestation and updates

OASIS provides a way to update a program while keeping the original program state. Intel SGX [57] improves enclave authentication by including initial page order and content in the measurement used for remote attestation. It improves OASIS's update mechanism by preventing software downgrades. SGX introduces the idea of having a dedicated signing enclave for remote attestation. Iso-X introduces the idea that adding pages updates the measurement of a compartment [43].

2.7.6 Enclave communication

Intel SGX allows enclaves to attest to each other by providing a trusted local key distribution mechanism [13]. This key distribution allows two enclaves on the same system to communicate through authenticated encryption. Sanctum also introduces the idea of mailboxes for trusted communication between enclaves [27]. Instead of protecting communication via cryptography, these mailboxes are secure by enforcing access control.

2.7.7 Memory partitioning

Sanctum [27] is resistant against cache and TLB side-channel attacks. Sanctum does this by flushing the first-level cache and TLB during a context switch, having a dedicated partition in DRAM for each enclave and using a cache coloring technique to avoid collisions between trusted DRAM partitions and untrusted partitions in the LLC. Other work has adopted this partitioning approach [17, 58]. One of cache coloring's downsides is the static partitioning of the LLC, which carries a performance cost when running memory intensive programs and does not scale well to more than a few concurrent enclaves. However, for lightweight enclaves (see Section 2.6) this spatial restriction is not a big problem as long as normal applications can still run without these restrictions.

2.7.8 Virtual machines

Bastion ensures that when a different hypervisor is loaded, it cannot access the compartment data managed by the previous hypervisor [56]. Bastion is the first enclave system to implement a hypervisor that can run any conventional OS on top of it. AMD SEV similarly protects virtual machines as opposed to applications [14]. This thesis focusses on TEEs that run applications rather than virtual machines, but some security mechanisms apply to both.

2.7.9 Industry versus open source

Intel SGX is the first commercially implemented enclave solution [13]. AMD SEV-SNP [28] is an improvement on the original SEV [14], which adds memory integrity and side-channel protection to a commercial solution that isolates virtual machines. Keystone [58] is "the first open-source framework for building customized TEEs." It is noteworthy because it provides a full framework that can be used in future enclave research. Arm realm management extension [15] shows how an enclave system can coexist with and complement an older TEE technology like TrustZone [9]. Industry, research and open source solutions are all necessary to further the field of enclave execution environments.

2.7.10 Trusted computing base

Iso-X [43] is a hardware-only solution without software in the TCB. Most other solutions have some form of trusted firmware so that there is a balance between the hardware and software complexity. Usually the goal is to make the TCB as small as possible to gain confidence in that it operates correctly. Alternatively, formal proofs can be used to prove security properties of the software [11] and hardware [59].

2.7.11 Multithreading

Sanctum [27] is the first solution to allow enclave multithreading. Sanctum does not partition the first-level cache, so the core can only use hyper-threading with threads from the same enclave. AMD SEV-SNP [28] also allows simultaneous multithreading and speculation to be disabled or restricted. Most other systems disallow simultaneous multithreading to avoid side channels that are possible through sharing resources like execution units.

2.7.12 Intra-core side-channel protection

MI6 [17] protects enclaves running on the same core by disallowing simultaneous multithreading and introducing a purge instruction. It is the first solution to expand the scope of protection against contention-based attacks in the full memory hierarchy. Previous works focused on cache contention, meaning that contention in other parts of the memory hierarchy still leaks data. For example, contention based on cache request buffers or DRAM request buffers. The resulting requirement for the DRAM controller to be constant time increases average DRAM access time. MI6 also has similar downsides to Sanctum as it uses an LLC partitioning scheme that does not scale. AMD SEV-SNP [28] introduces the idea of partitioning the architectural resources like the branch target buffer, debug registers and interrupt handling. However, it still lacks protections against other architectural side-channel attacks. Arm realm management extension [15] requires developers to make use of the newly added side-channel and Spectre mitigation features in the instruction set architecture (ISA), such as "consumption of speculative data" or "virtual and physical speculative store bypass" barriers [60], to protect against these transient attacks. Using these barriers seems to be the only way to add some form of protection for intra-core side-channel attacks on a conventional system. We can see in MI6 that a global solution for the whole system does affect the performance of normal applications that have a high memory footprint. However, if we are running enclaves often, each context switch would need one of these barriers and would degrade performance of regular applications, which could discourage the use of enclaves altogether.

2.7.13 Threat model divergence

Threat models in the enclave design space need to be standardized because different enclave solutions claim to protect against different types of attackers. Even different versions of the same platform have different protections. This stifles widespread adoption of enclave systems, as developers of multi-platform systems are forced to learn the intricate details of each enclave system they use before they are able to port their security critical applications to that platform. This, in turn, leads to a stagnation in the field of security, since developing a new system itself has the overhead of developing a whole new threat model to make progress. Creating a whole new threat model also makes recruiting new developers more time consuming.

To standardize a threat model, I consider the differences between modern enclave systems. At a high-level, most enclave systems protect against similar attacks; they agree that enclave code and data must be protected for confidentiality and integrity, and that these protections need to hold up against a privileged attacker (e.g. the OS kernel). However, a significant divergence becomes apparent through a more detailed analysis, for example:

- The first version of AMD SEV [14] only protects against attackers that observe and not tamper.
- Intel SGX [13] and Timber-V [29] do not protect against attackers that exploit side channels.
- Sanctum [27] does not protect against attackers that exploit side channels that are not based on cache line contention.
- MI6 [17] does not protect against attackers that run simultaneously on the same processor.
- Smart cards [21] protect against attackers that perform side-channel attacks on any level, including analogue measurements that require physical access [61].

The primary source of divergence is in coverage of side-channel attacks. Table 2.1 shows the coverage of side-channel attacks for previous enclave systems. The “proposed” column is the threat model that is presented in this chapter and is described in more detail in Section 2.8. Divergent threat models cause no two enclave systems to be the same.

2.7.14 Physical isolation

In the future, it may be possible to prove resistance to transient execution attacks in high performance cores. Formally specifying what properties are needed to achieve this has been a useful recent step [69]. However, this technology is not mature enough yet to use in current multi-core processors.

Smart card technology [21] uses physical isolation instead to prevent exploitation of side channels. Introducing a physical separate chip reduces the number of shared resources and therefore significantly decreases the attack surface for side-channel attacks. However, for fast communication and performance, sharing of the memory hierarchy is useful. Smart cards also include physical attacks in their attacker model, which includes clearing data upon drilling, freezing, etching and being resistant to power analysis among other attacks. Since enclave

	Side-Channel Attack	*Timber-V [29]*	*Intel SGX [13]*	*AMD SEV-SNP [28]*	*Sanctum [27]*	*MI6 [17]*	*Proposed [1]*
Inter-core	Cache line [27]	○	○	○	•	•	•
	Cache request [17]	○	○	○	○	•	•
	Control flow leakage [62]	○	○	○	○	○	○
	Denial of Service [63]	○	○	○	○	○	○
	DRAM row [30]	○	○	○	○	•	•
	DRAM request [17]	○	○	○	○	•	•
	DRAM bit leakage [64]	○	•	•	○	○	•
Intra-core	TLB [35]	○	○	○	•	•	▨ •
	Execution unit [36, 37]	○	○	•	○	•	▨ •
	Branch predictor [39]	○	○	•	○	•	▨ •
	Speculative execution [16]	○	○	•	○	•	▨ •
	Hardware accelerator [65]	○	○	○	○	○	▨ •
Physical	DRAM cold boot [66]	○	•	•	○	○	○
	Power analysis [67]	○	○	○	○	○	○
	Chip attacks [68]	○	○	○	○	○	○

Table legend:

○	Protection is *not* provided
•	Protection is provided
▨ •	Protection is provided by physical isolation

Table 2.1: A comparison of enclave threat models by inter- and intra-core side-channel attacks covered, as well as physical attacks. I published an earlier version of this table in 2020 [1].

threat models focus on protecting against privileged software these protections are not necessary.

Other technologies that use physical isolation include Apple's Secure Enclave [70], Microsoft Pluton [71] and Arm Corstone-700 [72]. Apple's Secure Enclave is a physically separate computational units on the same system on chip (SoC) as the main application cores. It only runs vendor trusted code, which is fundamentally different from the enclave model that assumes each enclave is mutually distrusting. Microsoft Pluton is similar to Apple's Secure Enclave, but for Windows devices [71]. The idea is that it can do secure boot, runtime verification and hold sensitive data necessary for biometric authentication. Arm Corstone-700 also has what it calls a "secure enclave" but just like with Apple's Secure Enclave, their notion of an enclave is different from what I call and enclave in this thesis: I call an enclave an environment similar to what Intel SGX implements, where any user application can spin off a secure compartment and the enclave system must protect enclaves from each other. Arm themselves call what I call an enclave a "realm" and they have released the Arm realm management extension [15], which allows multiple mutually distrusting compartments to exist alongside each other.

In essence smart cards or other physically isolated processors do so to protect sensitive data from attacks such as side-channel attacks. They recognize that physical isolation is one of the best defenses we have against them. There are many security systems out there that use physical isolation for security, however, specifically using these physically isolated cores to run mutually distrusting code as is the case with enclaves is something that has not been done before.

Physical isolation splits the space of side channels into intra- and inter-core attacks. Intra-core side-channel attacks require the attacker and victim to run on the same core, while inter-core side-channel attacks lack this requirement. Table 2.1 shows that physical isolation protects against all of the intra-core side-channel attacks. Physical isolation protects against intra-core side-channel attacks by preventing spatial multiplexing, but it does not automatically protect against temporal multiplexing. For example if an enclave needs to run on a core that another enclave is currently running on, then a secure context switch needs to make sure to scrub all microarchitectural state of the previous enclave. Nevertheless, physical isolation is a powerful tool to prevent a whole class of side-channel attacks on enclaves.

In terms of inter-core side-channel attacks, previous physically isolated systems would just have their own memory. Even in systems where the secure cores can access the insecure memory, they still have their own memory hierarchy to prevent inter-core side-channel attacks like those based on the cache (e.g. Apple's Secure Enclave and Microsoft's Pluton). The main difference between previous physically isolated systems and one that can run enclaves is that there are now mutually distrusting pieces of code running in the secure context. Also, it is desirable to give system designers the flexibility to use memory and caches for secure and insecure cases to save area and improve communication latency.

2.8 Threat model

Note: This section is based on a paper published by the author of this thesis [1].

The foundation for this threat model is protection against attacks that can be launched from privileged software, which is similar to that of previous enclave systems [13, 17, 27]. Enclave systems generally protect the following assets from attacks launched by privileged software and other enclaves:

- Confidentiality of memory: secret content does leak across enclave boundaries.
- Integrity of memory: code and data cannot be tampered with from outside an enclave boundary.
- Authenticity of the enclave system and enclave: an enclave may attest that it is running on and bound to a secure environment.

This means that enclave threat models include *direct attacks* like reading from and writing to memory, which can be launched by an OS or a malicious enclave. These attacks also include writing to memory via a direct memory access (DMA) request. For example it is trivial for privileged software on the central processing unit (CPU) to use the graphical processing unit (GPU) to make a DMA request.

It is also important to be precise about what is *excluded* in the threat model:

- Enclaves that leak sensitive data by having secret-dependent control-flow.
- Attacks that require physical access to the machine – for example to measure analogue properties, like power, voltage and sound.
- Bugs in the trusted part of the hardware or in the TCB.
- Denial of service attacks.

Additionally, I assume that an enclave system adhering to this threat model lack sensors that can measure physical properties, like power, to a significant resolution. If these sensors do exist then these should be inaccessible by software or otherwise excluded from the attacker model. One limitation of this assumption is that most modern systems have temperature sensors and microphones, but this is left to future research to guarantee that these side channels cannot be exploited by a privileged, software-based attacker.

Varying execution time on an architectural level can lead to a side-channel attack. With this side channel the attacker measures the execution time of the victim, and if the victim's control flow depends on data values, then this leaks information. There is nothing the micro-architecture can do to eliminate side channels that are present on an architectural level. For this system I consider architectural side channels out of scope, since it should be an enclave's responsibility not to leak secret data in such a manner.

All of the above is similar to what previous solutions have considered, but there is a lack of consensus on protection against side-channel attacks. My threat model includes *side-channel attacks* that can be launched from privileged software; like timing of memory requests and contention of execution units. To motivate this choice of threat model, I look at previous solutions and their inconsistent coverage of side-channel attacks. These findings are summarized in Table 2.1 with the "proposed" column presenting my threat model. This threat model is different from other threat models by aiming to protect enclaves from attacks including side-channel attacks by a privileged, software-based attacker. The system discussed in the rest of this thesis assumes the attacker and threat model described in this section.

2.9 Verifying security properties

Previous work has taken different approaches to verifying security properties [73]. For example, there are testing frameworks as well as simulation and formal methods. In industry certification schemes and standards are used for assessing levels of security. For example FIPS [74] and Common Criteria [75] both have different standards and different levels at which they can certify products.

2.9.1 Testing frameworks

Testing frameworks can be run on any processor that share the same ISA. In essence testing measures controllability, which finds whether a system produces the correct output for a given input. Some of these testing frameworks are automated verification tools that can discover potential transient side channels [76–81]. There are also random sequence generators with interactive deepening that find counterexamples for RISC-V compliance [82]. Although usually used for generic testing, these generators can also verify security properties by comparing an implementation to a golden model.

2.9.2 Simulation

Instead of measuring controllability, simulations measure observability, where the output of the simulation is used to infer how the internals of the system work. Specifically I am interested in simulators for side channels

There has been some work on modeling cache side channels based on just the cache parameters [83]. Using these parameters they create a mathematical model of the cache and use simulations to model the side-channel leakage. Another piece of work uses finite state machines to detect residual interference with cache side-channel mitigations [84]. Simulation has also been used to verify that programs are not vulnerable to speculative execution attacks [69] or side-channel attacks [85] on any hardware implementation.

2.9.3 Formal verification

Formally defining an abstract machine and what side-channel attacks look like on them has also been done before [59]. Security properties can be proven on the abstract machine and implementations can then be proven equivalent to the abstract machine. The way these formal proofs work is that they prove invariant properties, in this case the confidentiality and integrity of enclave data. Side channels such as those through caches have been modeled formally before and systems can be proven secure against them [59]. The proof is still only as good as the formal model of the channel and will only be useful for known side channels. Similar to formal proofs on hardware, software mitigations can also be proven to be secure, like masking in block cipher implementations [86, 87].

At the moment proofs are possible on an abstract model of a processor or by looking just at a software implementation. To date this has not been used to prove a modern application processor secure, and it is not clear whether this will be practical in the near future. In the meantime simulation based approaches remain a useful approach to investigate and argue about side channels.

2.10 Research challenges

Previous work has gone in two directions: adding enclave functionality to existing processors [13, 27] and adding security co-processors for a completely isolated environment [9, 21]. Adding enclave functionality to existing processors has the benefit that enclaves can run on high performance cores, and that resource management can still be left to the untrusted OS. The challenge with this is securing enclaves from side-channel attacks that are launched from privileged code. Adding security co-processors has the benefit of being protected against intra-core side-channel attacks because the security critical software has its own resources.

The challenge with these systems is that usually they are a walled garden where only vendor approved software can run and all the complexity of resource management must now be duplicated in the secure environment. The question is how we keep allowing any application to spin up an enclave, keeping most of the complexity of resource management in the untrusted OS and having the same level of protection against attacks from privileged code that dedicated security co-processors have.

Based on the use cases that I discuss in Section 2.6, it would make sense for these applications not to want to trust the OS they run on top of. With secure remote execution and keeping application secrets, confidentiality and integrity of enclaves need to be guaranteed. If a developer is willing to trust a cloud provider and OS producer, then they may not need to use an enclave system at all, but if they have decided that they want to run their application in an enclave it is unlikely that it would be acceptable to be exposed to side-channel attacks of any significant bandwidth. For the public ledger use case, it is undesirable to trust the OS since that is in the user's control who has a financial incentive to modify the public ledger. The main question that can be raised is whether this use case will ever be used in real life, but if it is then it will definitely need protection against all attacks from privileged code including side-channel attacks. In these use cases it is also undesirable and unrealistic for these enclaves to always be vendor vetted, approved and trusted. A system is necessary that allows third-party applications to securely run an enclave in a mutually distrusting environment.

One approach to provide this protection is by taking existing enclave systems and hardening them against intra-core side-channel attacks. To a certain extent non-enclave processors need to be hardened against these side channels anyways. However, the difficulty here is that the threat model of an enclave is different from a regular application. A regular application trusts the OS it is running on and so it only needs protection from other applications that are running on the system. An enclave on the other hand also needs protection from the OS and other privileged code on the system. We can ask the following questions: How much is the cost of adding this extra security requirement? Will it have a significant impact on performance? Is it even feasible to implement this given the scale and complexity of modern processors?

The system that is proposed in Chapter 4 aims to take the security co-processor approach, but add the convenience of an enclave system to it. The secure cores in this system would not be restricted to run vendor approved software. The system also adds critical enclave functionality like remote attestation, allowing the untrusted OS to do most of the resource management and a convenient application programming interface (API) for developers. A number of challenges and questions come to mind in such a system: Can a system like this co-exist with existing processors without slowing them down? How do we protect against inter-core side-channel attacks? What would the area overhead of these extra processors be? Is it possible to make this system secure without the walled garden approach? How to perform a context switch between two enclaves securely? How many enclave cores are necessary? How do we secure the memory hierarchy against direct and side-channel attacks? Do we need encryption or can we use memory tagging? How do we prevent collision and contention attacks in the cache hierarchy?

To be clear, the former approach is valid and evidenced by the previous enclave systems that have been developed in this paradigm. The goal of this research is to explore an alternative paradigm and determine whether an enclave system like this can work. Not all of the questions in this section are answered in this thesis, but the aim is to show an initial implementation and determine whether it is a viable alternative to the current paradigm.

2.11 Summary

In this chapter I discuss previous enclave systems in terms of their contributions to the field as well as what type of threat models they assume. I present a threat model that builds on previous work and discuss how physical isolation is a promising idea to protect enclaves against intra-core side-channel attacks. All of this sets the stage for discussing side channels in more detail and looking at my newly designed enclave system.

Chapter 3

Transforming side channels into direct channels

3.1 Introduction

In this chapter I present a new methodology that converts side channels into direct channels to make a security analysis. Developing a direct channel that corresponds to a side-channel attack is done by analyzing the basic primitives of the attack, directly exposing these primitives in a simulator and removing the confounding variables to reduce noise. A set of transmitter and receiver programs can then communicate over this direct channel. Finally, mitigations can be proposed to close the side channel. Once a new side channel is discovered, my methodology provides an easy platform to prototype and analyze such attacks. In the same simulator, all mitigations can be implemented before having to implement a complete system, but this step is optional since distilling side channels down to their basic primitives usually means that the mitigation trivially closes the channel.

The main contribution of this chapter is to enhance the engineering practices that are usually based on human intuition and hand picked analysis. The methodology gives enough structure to the analysis that it enhances the understanding of the side channels that are deemed important, but it does not require the more laborious step to go to formal analysis or analyzing a whole system.

The ultimate goal is to understand the complete set of side channels that fall within a given threat model. To achieve this I convert these side channels into direct channels of communication on a simulator. This means side channels can be analyzed in the same manner as direct channels like storing to and loading from memory. With each side channel having a corresponding direct channel, the set of direct channels of communication becomes a superset of all the known side channels.

3.2 Methodology

When designing a security sensitive system, architects currently employ the following process: they decide on a threat model, enumerate the attacks that are encompassed by this model, choose a set of measures to mitigate these attacks, implement a system and then reason qualitatively about its security. My proposal for turning side channels into direct channels allows designers to understand important side channels better and make an informed decision about which mitigations to implement. In more detail, system designers should take these steps for each side channel before implementing a system with a new threat model:

1. Analyze the basic primitives on which the side channel depends.

2. Expose these basic primitives as a direct channel in a simulator.

3. Remove confounding factors that create noise on the channel.

4. Implement a transmitter and a receiver to communicate over this channel.

5. *Optional*: choose candidate mitigations for the attack and evaluate whether they close the channel.

Previous methodologies to verify security properties are described in Section 2.9. There are specific tools to detect vulnerabilities such as transient execution attacks, and formal methods. At the moment verifying attacks on real system is a cumbersome task, so simplifying this process is desirable. Formal methods are great, but as of yet there is no way to formally verify a complete, realistic hardware system. Certification schemes are used in real world systems to make guarantees on security. My methodology could help in the certification process to better understand attacks. Previous simulation based approaches already require a deep understanding of how a side channel works and the different ways in which it manifests within the complexity of modern systems. My methodology describes the steps to understand the side channel on a conceptional and fundamental level first, and this can be used as a stepping stone to model the channel in more detail. The main point of the methodology is to structurally approach the initial analysis of side-channel attacks. What is usually a manual process based on human reasoning and published attacks can now follow a predictable recipe to better understand the important side channels in a threat model.

3.3 Existing engineering practices

Existing research practices try to verify security properties in certain ways, including human-based reasoning, taint tracking, testing frameworks and formal proofs. Most of these are described in Section 2.9. Human-based reasoning is when engineers reason about all the side channels they know of and devise new or existing countermeasures that can be included in their processors. My process is mainly aimed at enhancing this way of doing things, but there are also alternatives.

Taint tracking is a way to model how information flows through a processor [88]. The way it works is that if a piece of information is used to calculate or influence something else then that result is labeled as tainted. This requires special changes to the processor and it can be difficult to have enough colors and stop everything in the processor from becoming tainted. Another approach that avoids this is using testing frameworks, where known tests are done on a processor to see if it leaks information [76–81]. This takes less instrumentation in the processor but it is much more difficult to reason about completeness.

An ideal solution is to *prove* that a processor does not leak information through side channels, similar to those that have been done on abstract models of processors [59]. Once it is possible to create a high-performance core with this type of guarantee, it will probably supersede any existing methodologies. Until that is possible, my methodology can help make the human-based engineering practices more structured and rigorous. To show how this methodology works, I sketch how it can be applied to known side channels.

3.4 Inter-core side channels

Inter-core side-channel attacks are those that can be launched between processes running on separate cores. In this section I show how my methodology applies to two such attacks: cache line and dynamic random-access memory (DRAM) row buffer collisions.

3.4.1 Cache collisions

For this type of attack, the cache's associativity and eviction policy can both act as confounding factors, potentially leading to noise on the channel. Other confounding factors can include instruction fetching, which can pollute the cache in unexpected ways leading to unwanted behavior from the point of view of an attacker, or nondeterministic memory hierarchy behavior which can lead to mistaking a miss for a hit or vice versa. Another confounding factor is when there are other threads running on the system that cause an eviction, which leads to a false positive during the probing part of the attack.

To directly expose priming the cache, I use a direct-mapped cache instead of a set-associative cache so that one load instruction is guaranteed to evict a cache line. To directly expose the probing part of the attack, I introduce a control and status register (CSR) that contains the number of cache misses per core. This avoids having to time a load to determine whether a miss has happened. To have the same power as the original attack, a miss count per cache

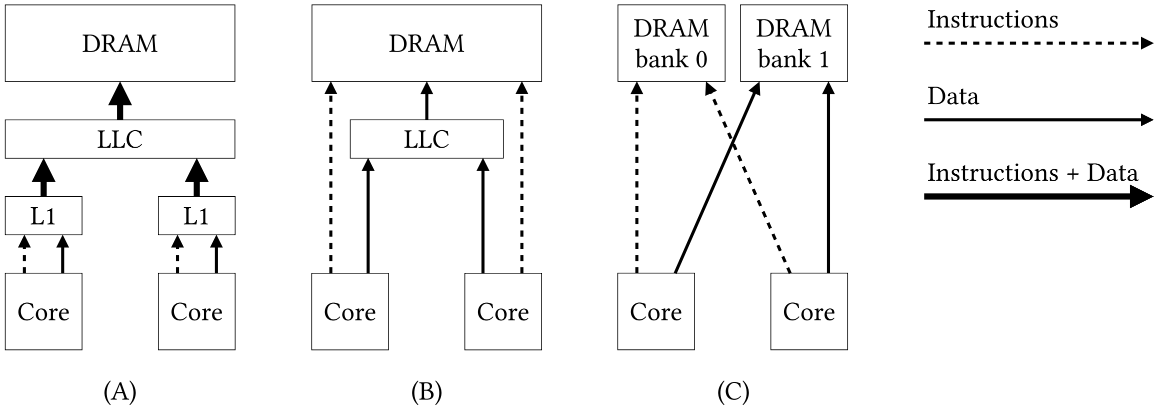

Figure 3.1: Three diagrams showing cores, first-level (L1) caches, LLC, DRAM and how they are connected. A conventional memory hierarchy (A), the memory hierarchy that is used in the cache collisions experiment (B) and the memory hierarchy that is used in the DRAM row buffer collisions experiment (C).

line is necessary, but the principle on how the transmitter and receiver operate is the same in either case.

Other confounding factors in this experiment are removed by simplifying the memory hierarchy. Figure 3.1 (A) shows a conventional memory hierarchy and Figure 3.1 (B) shows the changes that are made for this experiment: removing the first-level caches makes the LLC handle all memory requests and ignoring instruction fetch requests in the LLC makes the miss count only reflect data accesses.

I implement this modified memory hierarchy and the direct channel in an instruction-level simulator based on Spike. Afterward I create a transmitter and a receiver that run in parallel on two separate cores. To avoid having to synchronize the transmitter and receiver, I configure the simulator to make the two cores run in lockstep. Figure 3.2 shows the pseudocode of the transmitter and Figure 3.3 shows the pseudocode of the receiver, which together take advantage of the direct channel in my simulator. To keep the transmitter and receiver code simple, I make their for loops take equal time. The colliding addresses in the transmitter and receiver are different addresses, but are mapped onto the same cache line. The transmitter runs through each bit of the value *s* and performs a load depending on the bit's value (Line 5 in Figure 3.2). This means that Line 6 in Figure 3.3 causes a miss depending on whether the transmitter does a load or not. Line 5 and 7 in Figure 3.3 reference the miss count of the LLC. The receiver uses the difference between these two miss counts to decide whether to set the bit in *s* or not (Line 9 in Figure 3.3). Having removed the confounding factors and exposing a direct channel makes this code concise and efficient. Figure 3.2 and 3.3 are pseudocode representations of RISC-V assembly. The real assembly transmits one bit per 11 RISC-V instructions.

```
Result: Secret is sent to receiver, which is listed in Figure 3.3
a ← colliding_address;
s ← secret_value;
for i ← 0 to 31 do
    if $\left\lfloor \frac{s}{2^i} \right\rfloor \bmod 2 = 1$ then
        load_memory(a);
    end
end
```

Figure 3.2: Transmitter for memory collision channels (originally in RISC-V assembly).

```
Result: Secret ends up in s
a ← colliding_address;
load_memory(a);
s ← 0;
for i ← 0 to 31 do
    m ← miss_count();
    load_memory(a);
    m ← miss_count() − m;
    if m > 0 then
        s ← s + $2^i$;
    end
end
```

Figure 3.3: Receiver for memory collision channels (originally in RISC-V assembly).

Finally, I choose a set of mitigations to this attack and implement them in the simulator. Costan [27] proposes a static cache-partitioning scheme and Wang [89] proposes a flexible cache-partitioning scheme to mitigate this attack. I ignore random invalidation or other statistical mitigations in this analysis, but I discuss them in Section 3.7.5. Spike has a simple cache simulator, which I adjust to implement different partitioning schemes and see which scheme closes the channel. In my results both proposals mitigate the priming part of the attack and stop the communication over the channel.

In this experiment I introduce a new direct channel in the simulator based on how cache collisions work, and this direct channel is then used to evaluate proposed mitigations.Both proposed mitigations close the direct channel, and hence both are suitable mitigations for the original attack. All of this can help provide security analyses of systems such as trusted execution environments (TEEs).

3.4.2 Dram row buffer collisions

To show the benefit of this new methodology, I choose DRAM row buffer collisions as another example of a side-channel attack. Although this attack focuses on a different part of the memory hierarchy, the elegance of this methodology becomes apparent because I can reuse a significant portion of my previous analysis. The basic primitives of this attack are accessing a different row on the same bank as a victim and detecting whether the victim has re-opened a row on that bank. These basic primitives are similar to the ones for cache collisions because, in essence, this attack is a "prime and probe" attack on the row buffer instead of a cache line.

This realization makes the next steps easier to execute: I expose the row miss count instead of the cache miss count and I remove all caches instead of just the first-level caches. Figure 3.1 (C) shows the adjusted memory hierarchy for exploiting DRAM row buffer collisions and how it compares to the cache collision experiment. All instruction fetches are directed to a different bank from the data, which removes noise from the measurement. These changes greatly simplify using this channel for communication.

The transmitter and receiver described by Figure 3.2 and 3.3 can be reused in this case. The colliding addresses in variable *a* must now be located on the same bank but in a different row rather than being mapped onto the same cache line.

To close this channel, the DRAM controller can be made to take constant time [17]. The probing part of the attack would be blocked because closed rows are now accessed in the same amount of time as open rows. Accessing open rows would take longer: in one of Samsung's DDR4 DRAM chips this increases access time to already opened rows by between 40% and 50% [31].

Because of the similarities with the cache collisions attack, I compare the mitigations of these two attacks and whether they can be applied to the other. Having constant access time would also stop the cache collision attacks, but would defeat the purpose of caching. Caching allows a subset of memory to be accessed more quickly than interacting with memory directly. Going the other way, DRAM can be partitioned on a per-bank granularity. Memory management would be more complicated in this scenario and there are limited number of banks for partitions to be created on. The elegance of the new methodology allows system designers to start seeing these parallels and reason about why certain mitigations are used in one case while not in another.

Studying this second example shows that my methodology can expose redundancy in analysis of side channels. This methodology allows for equal and quantitative treatment of all known side-channel attacks that are important to a threat model.

3.4.3 Other inter-core side channels

After doing two full examples, I now describe a number of common side-channel attacks for which I limit discussion to analysis of the basic primitives, propose how to expose them through a direct channel and list mitigations for these attacks.

Cache requests

Depending on how many cache requests are currently outstanding, the time it take to serve these requests will vary. The basic primitives of this attack are accessing the same buffer as a victim to make cache requests and detecting whether there is any contention with other requests. To directly expose the contention, a cache arbiter can increase a counter every time a core makes a request while another request is being served.

Dram requests

DRAM request contention is similar to cache contention, but instead of contention happening between the core and the cache, it now happens between the cache and DRAM. The back pressure created by DRAM contention can cause the cache request buffers to become full and leak information across domains [17]. The basic primitives are sharing a channel for making memory requests to DRAM and detecting whether a victim is making a concurrent DRAM request.

DRAM bit leakage

DRAM bit leakage is based on the physical implementation of DRAM chips and relies on slight changes in charge to adjacent rows when performing a DRAM write, which can cause bit flips to occur [32]. The basic primitives are finding a row adjacent to a victim's row, affecting values in a row by writing to adjacent rows and fooling the victim into using the tampered value. To make this a direct channel, a special store instruction can write to adjacent rows in the same bank.

3.5 Intra-core side channels

As opposed to inter-core side channels, intra-core side channels require both the victim and attacker to be collocated on the same core. This section shows how my new methodology can be used on example intra-core side channels.

3.5.1 Page access tracking

Page access tracking is a side channel that is performed by privileged software, which is usually in charge of handling page faults [35]. These page faults leak information on which pages are used by a victim. Page faults occur when a memory translation is unavailable in the translation look-aside buffer (TLB), which is essentially a cache of recent translations from virtual to physical memory. The basic primitives of this attack are the ability to evict an entry from the TLB and to detect when the victim is accessing this entry again. Exception handling already gives the attacker a direct handle to knowing when a TLB miss happens and evicting TLB entries can be directly exposed. Similar attacks are possible by monitoring the "access" and "dirty" bits in the page table.

3.5.2 Execution unit usage

Execution unit contention is a side channel created from simultaneous multithreading. When multiple threads are executing on the same core, they share many resources like execution units and micro-op caches [38]. This sharing allows contention to occur between instructions from the attacker and the victim. The basic primitives of this attack are sharing the same execution unit as a victim and detecting whether a victim is using it at the same time. Creating a direct channel can be done by exposing counters per thread and per execution unit that increment every time contention occurs on an execution unit.

3.5.3 Branch shadowing

Branch shadowing uses the shared nature of the branch predictor to extract information about a victim's control flow: whether a victim's branch is taken affects the way branches in the attacker's code are predicted [39–41]. This attack relies on a few of basic primitives: finding an address that maps onto the same branch predictor entry as a victim's branch, and detecting whether a victim's branch was taken or not based on that branch predictor entry. To create a direct channel, a shadow branch predictor can hold the taken status of the most recent branch that used each entry.

3.6 Direct channel classification

In this chapter I discuss eight different side channels and how to convert them into direct channels. In general these direct channels fall into two classes. The first class is based on *exposing new microarchitectural statistics*, which previously had to be accessed indirectly. Examples of these statistics are counts of misses in the memory hierarchy and contention occurrences on shared resources. The second class of direct channels is based on *exposing existing microarchitectural state* directly. Examples of exposing this state are giving direct write access to adjacent DRAM rows, the ability to evict TLB entries and direct read access to the taken status of each branch predictor entry. Exposing a combination of microarchitectural statistics and existing microarchitectural state allows any software-based side channel to be turned into a direct channel.

Through this analysis I show how system architects can apply this new methodology to existing and newly discovered side-channel attacks. For each attack, architects identify basic primitives and directly expose them in a simulator. Then architects remove confounding factors and implement a transmitter/receiver pair to use this new channel for communication. Architects can use this simulation to find out which candidate mitigations close the channel, for example cache partitioning stops "prime and probe" attacks (see Section 3.4.1).

3.7 Discussion

To explore the context in which this new methodology can be used, I discuss its impact in terms of improving the design of TEEs and trusted software. I also highlight the relationship between my simulation methodology and specific implementations. I discuss possible extensions to this work such as exploring speculative execution in more detail and characterizing noise.

3.7.1 Trusted execution environments

Using my methodology in the design of TEEs can help researchers and developers make informed choices about which attacks to include in their threat model, and to better understand the side channel attacks that they choose to include.

Taking MI6 as an example, they state that their "isolation mechanisms exclusively address software attacks" [17]. They also do not protect against denial of service attacks. Finally they explicitly mention that they ignore attacks like DRAM bit flipping [17]. Afterward the paper goes directly into what mitigations they use to guarantee protection against side-channel attacks [17]. I argue that it is valuable to apply my methodology before diving into the chosen mitigations.

Instead of describing their chosen security measures, it would be better to use the threat model to create a list of relevant side-channel attacks. Once that is done, researchers can apply the new methodology to understand each side channel better before considering which mitigations to implement. In MI6 [17], they use mitigations ranging from sizing cache request buffers to avoid DRAM contention to disabling simultaneous multithreading to avoid execution unit contention.

This chapter applies the methodology of distilling the basic primitives of each side channel that is relevant for the enclave system that is described in the later chapters. The structured analysis, of how each side channel works and how a simple transmitter and receiver would operate, made it easier to organize the ideas of which mitigations to use. It also inspired the comparison of the threat models in Table 2.1, which then led to a set of mitigations to consider for Praesidio. Additionally, the implementation of the direct channels in Sections 3.4.1 and 3.4.2 are done in the same simulator as the implementation of Praesidio in Chapter 6. This means that we know it is possible to study these side channels in an instruction level simulator even if those usually only appear in real implementations.

3.7.2 Software development

Besides hardware development, software development can also profit from the analysis that my methodology promotes. Having examples of transmitter and receiver programs that are easy to understand helps developers identify which code constructions lead to information leakage. For example the pseudocode in Figure 3.2 and 3.3 shows developers what type of code makes it possible for an attacker to exploit collisions in the memory hierarchy. Taking this further, a developer can use a database of these example transmitters to argue that a piece of security critical code is unlikely to be vulnerable.

3.7.3 Speculation

Speculative execution attacks rely on two steps: influencing the speculative execution of a victim and using a side channel to extract information about this speculation. My methodology is useful in understanding and mitigating the side-channel aspect of this class of attacks. However it is still an open question whether speculative execution can be studied through simulation. For example system designers can use a window of instructions in which the simulator can look ahead to see what sequences of instructions are likely to be speculated. Designers can model speculative behavior in an instruction-level simulator by introducing the idea of branch prediction (variant 1), branch target prediction (variant 2), exception handling at commit (variant 3) and speculative store bypassing (variant 4) [16]. Designers can use this simulator to reason about security properties regardless of which speculative path the processor chooses. The main problem with this approach is that the number of possible speculative execution paths grows exponentially with the window size, a challenge left to future work. It would be interesting to see how this approach complements existing approaches on formally proving properties in speculation [69].

3.7.4 Simulation versus implementation

Side channels are inherently about the implementation details of computing systems. In this section I consider gem5 [90] and other microarchitectural simulators to be implementations because they simulate implementation details, while my simulator is more high-level and focuses on the minimum features necessary to study security properties. So why is simulation a good way to study something that is inherently dependent on implementation details? The simulation approach makes analyzing side channels simpler. It is also important to note that my methodology is not meant to replace implementing security systems, but to work in tandem with them.

Attempting to reproduce attacks on implementations might result in false negatives and a false sense of security since they often rely on many engineering details. My methodology makes studying attacks simpler, so that developers can make a more informed decision on which mitigations to select. This methodology is also more scalable because it allows studying classes of side channels on the simulator rather than having to develop a specific attack for each side channel on an implementation.

My methodology takes into consideration what the relevant side-channel threats are and how real implementations usually work. With this input it offers an efficient way to gain a greater understanding of known side channels and to select a set of promising mitigations to implement. This knowledge helps both in implementing real systems from scratch and in adjusting existing implementations. In essence this is a feedback loop where simulation provides ways to improve implementation and implementation provides information to improve simulation.

3.7.5 Characterizing noise

Another way in which my simulation approach complements implementation is in studying noise. In this methodology a system designer removes all noise from the channel. However, designers can still characterize noise. For example there are mitigations that increase noise over a channel by randomly invalidating parts of the cache [91]. Designers can take the signal-to-noise ratio of such a mitigation and simulate it on top of the relevant direct channels. Taking this noise into account, designers can adjust the transmitter and receiver pair to measure how the channel bandwidth decreases with the signal-to-noise ratio. This methodology allows designers to evaluate mitigations that create noise before committing to implementing a full system.

3.7.6 Limitations

Limitations of my methodology include that it only considers side channels that are already known, it might not be the best system to evaluate noise-based mitigations (see Section 3.7.5) and it does not provide the same guarantees as a formal proof. Even though determining the basic primitives of a side channel is essential in selecting appropriate mitigations, it is unclear whether the methodology can evaluate mitigations where it was not already clear that it would close the side channel. The methodology can be used to find mitigations that worked on other side channels that have similar basic primitives, but these suggestions are not guaranteed to have good performance. The methodology also works best in deciding how to completely close a channel. If this is not practical, then it might be better to estimate and minimize the amount of information that is leaked by measuring the side-channel vulnerability factor [92]. Mostly the methodology is a structured exercise that can enhance understanding of the side channels that are considered in a system and help identify mitigations that eliminate the channel, but it will not replace performance evaluations nor security proofs.

3.8 Summary

I present a new methodology for creating a security analysis of side-channel attacks by turning side channels into direct channels. Within this methodology, analysis for one side channel transfers to other side channels that rely on similar basic primitives. I show how known software-based side-channel attacks are translated by this new methodology and find that the direct channels are based on either exposing microarchitectural statistics or exposing existing microarchitectural state. The goal is to create a direct channel for each side channel so that the set of all direct channels in the simulator is a superset of all known side channels that are considered in a threat model. The analysis shows the plethora of attacks that are included in the threat model presented in Section 2.8 and what attacks I must protect against in my new enclave system presented in Chapter 4.

Chapter 4

Enclave system design

Note: Most of this chapter is based on a paper published by the author of this thesis [1].

4.1 Introduction

There are many classes of side-channel attacks including differential power analysis, fault injection using lasers and cache timing attacks. Since side-channel attacks that require physical access are less scalable, I choose to only protect against side-channel attacks that can be executed by software. Previous enclave systems do not fully protect against software-based side-channel attacks, which is where our new enclave system called Praesidio[1] comes in.

Software-based side channels can be classified into two categories: intra-core and inter-core channels. Intra-core side-channel attacks require the attacker and the victim to be co-located on the same core, while inter-core side-channel attacks lack this requirement. Of these two classes, intra-core side-channel attacks are the hardest to protect against, mainly due to the sheer number of shared microarchitectural resources in modern application cores.

To protect against all intra-core side-channel attacks, one class of trusted execution environments (TEEs) run sensitive code on dedicated hardware that is physically separate from the main processor. SIM cards, smart cards [21] and trusted platform modules [22] create a physically isolated execution environment to make security guarantees. My work applies this concept by running all enclaves on cores that are physically separate from application cores. This is different from previous enclave systems, which focus on adding enclave functionality to high-performance application cores. After physical isolation, the focus then becomes protecting against inter-core side-channel attacks that exist due to the sharing of common resources like the memory hierarchy. Protecting against inter-core side-channel attacks is still challenging, since side channels can be based on contention on cache lines, request buffers,

[1] Praesidio means “protection” in Latin, and it is the name of the new enclave system I propose.

buses, dynamic random-access memory (DRAM) row buffers, etc. However, the possibility of ignoring intra-core side-channel attacks reduces the number of channels that must be considered in a system.

Given that it is now the age of dark silicon [93], dedicated hardware can be introduced to perform key operations. For enclaves, this means that dedicated processors can be introduced that are designed to be highly robust against side-channel and speculative execution attacks. Having dedicated secure processors for enclaves means that the performance of fast application cores are unaffected by the enclave security requirements.

This chapter contributes to the field by designing a system that combines the power of physical isolation (smart cards, trusted platform modules, etc.) with the possibility that any application can run on a mutually distrusting enclave. The system design takes ideas from previous work, such as memory tagging, cache partitioning, enclave life cycles and more to create a complete system where enclaves are by default protected against intra-core side-channel attacks and by configuration protected against inter-core side-channel attacks. This chapter also shows that a physically isolated enclave system can be implemented without affecting the main application cores.

4.2 Related work

Praesidio sets itself apart by combining the benefits of physical isolation with the enclave execution model. I compare this work to previous systems in terms of threat model in Section 2.7.13. The physical isolation that is explored in Praesidio composes well with mitigations used in other enclave work like using physically unclonable functions (PUFs) [34], coloring the cache [27], purging the pipeline [17], etc. Physical isolation as it is used in trusted platform modules [22] and smart cards [21] is a powerful mitigation and using it for an enclave system is a contribution to the field. It is also decidedly different than the paradigm of making fast cores secure, which is what previous enclave systems have done [13, 15, 28, 58]. Physical isolation is a promising alternative because it avoids the costs of making fast cores more secure.

Memory protection in Praesidio is based on tagged memory. This is different from classic enclave systems which either use cryptography to enforce access control [13] or keep track of enclave ownership in a shadow page table [27]. Timber-V uses the idea of tagged memory to protect enclaves, but ignores side channels as part of the threat model [29]. Praesidio shows how tagged memory can be used in conjunction with physically isolated enclaves and this approach composes with existing optimizations like tag caches [52, 94].

Praesidio also introduces a user application programming interface (API) to interface with enclaves. Previous enclave APIs are based on the synchronous enclave model, where an enclave runs on the same core as the application [13, 17]. Praesidio provides a similar programming model to the synchronous interfaces by allowing shared memory for communication. The main difference is that enclaves run asynchronously to applications: they might run in parallel or if power is constrained the fast core might be suspended while the enclave is running on the secure core. Existing asynchronous interfaces for trusted execution environments like those based on the Global Platform specification [95] can be implemented on top of the functionality that Praesidio already provides.

4.3 System overview

Praesidio is an enclave system that enforces physical isolation while still providing the same functionality as conventional enclave systems and with minimal performance loss to fast cores. Praesidio enforces access control in the memory hierarchy without trusting the implementation of the application cores. This is possible because of the addition of secure cores which are designed to be highly robust against speculative execution and side-channel attacks.

Figure 4.1 shows the system overview of Praesidio, with software that runs on cores optimized for performance on the left and software that runs on cores optimized for isolation on the right. Any user application can launch an enclave through the user-level API by sending the initial code and data to the Linux driver. The driver tells the management shim to create an enclave and reserves memory from the operating system (OS) for the enclave. The driver prepares this memory before relinquishing control of these pages by donating them to the enclave through the management shim. During donation the management shim changes the access control of the page. The hardware is in charge of enforcing this access control: making sure that any enclave can only interact with its own pages, which is detailed in Section 4.4. The management shim also assists the driver in setting up pages for communication between the application and the enclave, as well as in scheduling enclaves. The enclave runtime offers functionality similar to the user API for enclaves, of which sending and receiving messages are the most commonly used.

Another way of looking at the system is from a hardware perspective. Figure 4.2 gives an overview of the additional hardware (in gray) that is required by Praesidio. It also shows that fast cores stay unmodified except for a memory interface that enforces access control on the communication between the core's private caches and the shared last-level cache (LLC). I discuss optional tag translation to reduce the LLC overhead in Section 6.5.2. Finally, a trusted boot read-only memory (ROM) is added so that the management shim can protect its pages and clear any sensitive data before untrusted code runs.

Though our system is built on Linux, the driver is simple and the enclave system is built in such a way that it should be composable with any rich OS. By rich OS, I mean any feature rich OS that provides lots of functionality for the applications that run on top of it but is therefore

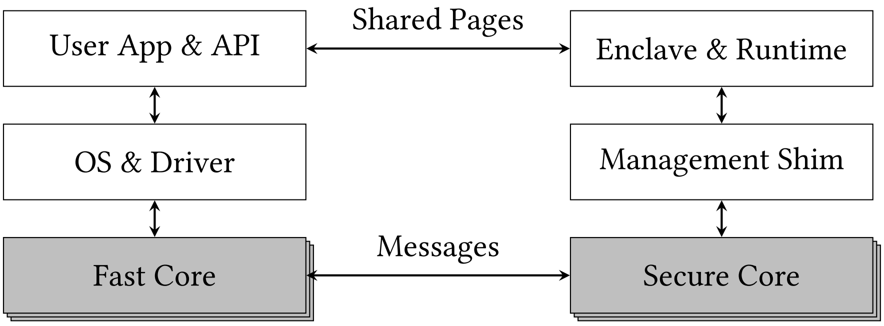

Figure 4.1: System overview of Praesidio. White boxes are software; gray boxes are hardware; arrows are channels of communication.

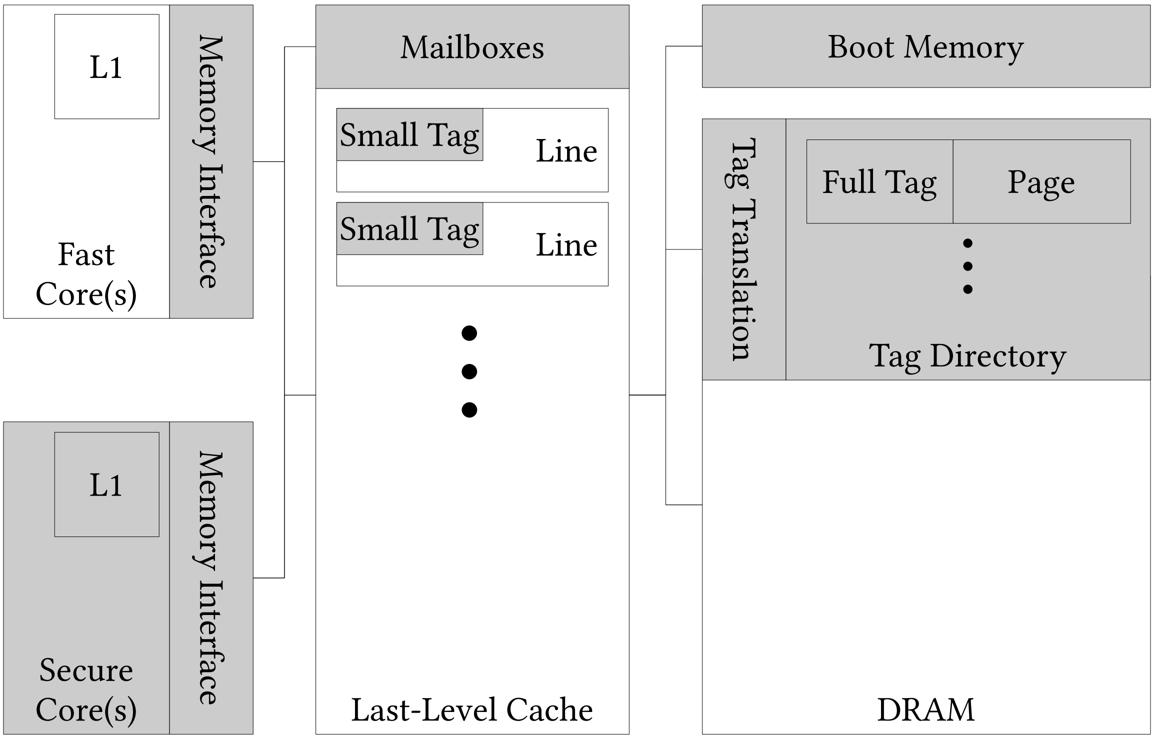

Figure 4.2: Hardware overview of Praesidio. The boxes with gray backgrounds represent additional hardware that is added by Praesidio. The boxes with white backgrounds are hardware that existed before adding Praesidio. The text L1 means first-level cache.

also more complicated and vulnerable to attacks. Examples of rich OSs are those based on Unix, Linux or DOS.

4.4 Memory protection

To protect enclaves against direct attacks, Praesidio tags each page with an enclave identifier. This is similar to physical address space tags used in the Arm realm management extension [15], except that Praesidio has a unique tag per enclave. Memory protection is enforced by the memory interfaces that are located between the cores and the shared cache, and by the tags that are stored in the LLC and the tag directory. The tags are managed by the management shim software. On every memory access that reaches the LLC, the tag of the memory is compared to the identifier of the currently running enclave. In this model normal applications and other software running on the fast cores are assigned the default enclave identifier. Loads and stores to a page that an enclave does not have access to are blocked by the memory interface. Loads return a known value, which is done by Intel SGX as well [13]. Blocking is acceptable in this case because illegal request only arise from an actively malicious process or from a process using a misconfigured page table mapping.

The tag directory is separate from the page table because tags protect physical pages, not virtual ones. In theory tags can be enforced in the page table of the enclave cores because the management shim is responsible for installing the memory mapping on those cores. However, the page logic of the fast cores is explicitly outside of the trusted hardware and software. This means that the tags must be enforced on the interface between the fast core's private caches and the LLC. For consistency I choose to keep tags separate from the page directory in the whole system.

To support communication Praesidio allows a page owner to add a reader tag to a page. This reader enclave can then read the content of the page, but is unable to modify it. Defaulting to one-way communication allows applications to follow the principle of least privilege [96] and provide only read access where that suffices. These one-way shared pages are used to implement ring buffers for communication, which is described in Section 4.5.7. Unidirectional communication between processes running in parallel is a well-understood concept due to the frequent use of Unix pipes. Where one-way communication is insufficient, applications can upgrade to two-way communication by setting up two communication pages in opposite directions. This way of doing communication is different from Intel SGX [13], which allows the enclave access to all of the applications memory, and from Sanctum [27], which has a dedicated in-memory mailbox system. Neither of these two systems work well with our physically isolated setup unless it is possible to run trusted code on fast cores, which is best avoided to limit the size of the trusted computing base (TCB) and modification of the fast cores.

The tag directory contains a mapping from physical pages to their associated tags. For each page there is an owner tag and an optional reader tag. The absence of a reader tag means that the page is private to the owner. Any page that is missing in the tag directory is considered to be owned by the rich OS, which operates under the default enclave identifier. Each line in the LLC is tagged with owner and optionally reader enclave identifiers. The following steps describe the way memory requests are handled in Praesidio and how to enforce access control:

1. The memory interfaces for both fast and secure cores ensure that each memory request is accompanied by the enclave identifier of the requester.

2. For memory requests from fast cores all code is considered to run under the default enclave identifier and so the content of the translation look-aside buffer (TLB), store buffer and first-level cache only contain entries of memory that the default enclave has access to. For secure cores the TLB, store buffer and first-level cache can either be private to each enclave or partitioned securely. When a context switch occurs on a secure core all of the local state related to that enclave must be flushed or partitioned securely. In both fast and secure cores any hit at this level of the memory hierarchy means that the memory request can succeed without checking any tags.

3. On a first-level cache miss and an LLC hit, the tag for the cache line of the LLC is compared to the requester's identifier as shown in Figure 4.3.

4. On an LLC miss, the request must first go through the tag directory before going to DRAM. The tag directory contains all mappings from (physical) pages to owner enclave identifiers and optionally reader enclave identifiers. The request is evaluated using the process shown in Figure 4.3.

5. If the request is allowed by the tag directory, the tag from the tag directory is optionally translated to a smaller version for in the LLC. For a discussion on this tag compression see Section 6.5.2.

4.4.1 Preventing inter-core side channels

Praesidio ensures protection against all intra-core side-channel attacks by segregating security domains onto their own cores. However, there are resources that are shared across cores, which introduce the possibility of inter-core side-channel attacks. Contention in shared caches, request buffers and DRAM are examples of inter-core side-channel attacks.

Cache line contention causes information leakage between security domains because one domain can evict a line from another domain. To solve this issue, previous work has introduced

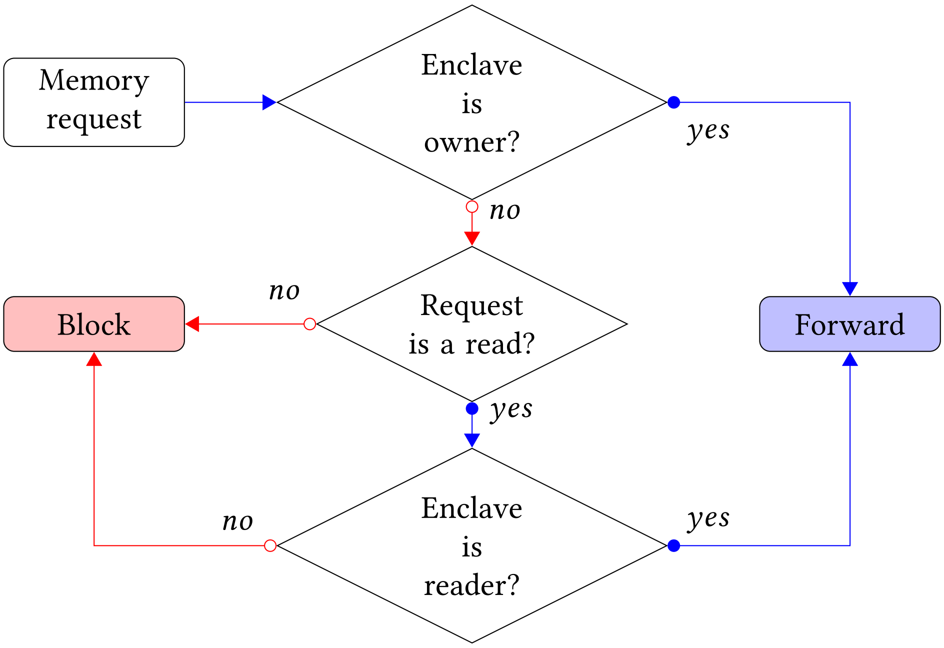

Figure 4.3: Process for checking whether a memory request is valid or not. This process is followed when checking whether a requester is allowed to access an LLC cache line or a page in DRAM.

static partitioning [17, 27], flexible partitioning [89, 97, 98] or noise [91]. Praesidio is configurable to use no partitioning scheme, a static partitioning scheme or a flexible partitioning scheme [89].

Because partitions are inherently assigned to different enclaves, there is no shared memory between any two enclaves except for when a reader is assigned to a piece of memory. Cache coherence is never a problem for any memory that lacks an enclave reader. This means that unless there is an explicit reader, the cache coherency mechanism will not cause any information flow between enclaves. It is worth noting that the side effects of the coherence chatter between two participants may be observed by other enclaves. My instruction-level simulator (Chapter 6) implements coherence, but lacks any observable latency based on caches. My hardware implementation (Chapter 7) disallows sharing of memory through caches. I leave it to future work to design a compartmentalized cache coherence protocol.

Inter-core side-channel attacks that rely on DRAM, like DRAM row buffer contention [30] or DRAM request buffer contention [17], are also important to protect against. The hardware overview in Figure 4.2 shows that DRAM is unaltered, so existing mitigations can be used for these problems. In Chapter 5, I discuss the security of Praesidio in more detail and how known inter-core side-channel attacks on enclaves can be mitigated.

4.4.2 Bootstrapping shim

At least one secure core must boot into the management shim so that the shim can start to manage the tags that protect memory. During this bootstrapping process all other cores must not interact with the memory until initialization is finished. The memory interface of each core can pause all memory requests while the management shim initializes. During initialization, the management shim clears any sensitive data from DRAM to avoid cold boot attack and tags all the pages that are owned by the management shim so that these cannot be accessed by other pieces of code. Bootstrapping trusted code at boot time is important to guarantee the integrity of the management shim code as well as protect the confidentiality of secrets like the platform key that is needed for attestation. This type of bootstrapping is quite common in mature security systems like trusted platform modules, which are used for trusted boot. The trusted platform module must run first to be able to guarantee the integrity of the booted OS.

4.5 Management shim

The management shim is the software part of Praesidio's TCB. The hardware enforces that pages can only be accessed by authorized enclaves, and the management shim is the only piece of code that is allowed to change the values in the tag directory. The management shim is not an OS and only implements the least amount of logic necessary to securely maintain enclave lifetime and transfer ownership of pages.

In Praesidio, along with most enclave systems, the resource management is still done by the rich OS, which has the benefit of leaving this complexity outside of the TCB. The management shim is only there to ensure that the OS follows the security rules. For example once a page is donated to an enclave, the OS cannot get it back unless it deletes the enclave. Upon deletion the management shim fills the corresponding pages with zeroes and gives the pages back to the rich OS.

4.5.1 Messaging

Communication in Praesidio is done via shared memory. I describe how pages can be tagged with a reader as well as an owner for communication in Section 4.4 and how this is exposed to the enclave in Section 4.5.7. To securely set up these pages and as a trusted way to manage enclave life cycles, Praesidio provides a messaging system meant to send infrequent, small messages.

I implement this messaging system by having a portion of the LLC that has a mailbox for each core (see Figure 4.2). A core can put a message into its own mailbox with a recipient identifier and it can check whether any of the other cores have posted a message for them.

Type	Arguments		Return values	
Create	–	–	Enclave ID	–
Donate	Enclave ID	Address	Success	–
Finalize	Enclave ID	–	–	–
Attest	Enclave ID	Challenge	Quote	Signature
Run	Enclave ID	Core ID	Success	–
Share	Page	–	Success	–
Delete	Enclave ID	–	Success	–

Table 4.1: Definition of different messages that issue requests from the OS driver to the management shim or share memory between enclaves. The challenge contains all the cryptographic information that the challenger provides for the enclave to create a valid quote.

The main purpose of this messaging system is for cross-core communication within the management shim and to allow the OS to send requests to the management shim. These mailboxes are memory mapped, so to write a message an enclave simply has to store to a special physical address. Since an enclave can only write to its own mailbox, it has to wait for the destination enclave to read the message. If that enclave is not scheduled, the sending enclave either has to wait for the management shim to schedule the destination enclave or discard the message by overwriting it.

Messages are sent from one enclave to another and the management shim has a dedicated set of enclave identifiers. The management shim has a unique enclave identifier per core, so that each instance of the management shim can be uniquely addressed for scheduling messages. Each message has a destination enclave identifier, a message type and a payload of maximum two arguments. The different message types are summarized in Table 4.1. The return values in the table are themselves messages in the opposite direction.

4.5.2 Enclave life cycle

Sending messages to the management shim allows an OS to manage enclave life cycles, while the management shim upholds the security requirements. Figure 4.4 shows the different states that an enclave can be in and which messages cause the state transitions. This state diagram is similar to that of previous work by Costan [27], but with explicit building and error states. The management shim keeps track of which state the enclave is in through an enclave data entry. This means that the management shim can enforce security properties; for example no pages can be donated to an enclave that has already run.

An enclave data entry starts out being empty and goes to the created state when a “create” message is received. Once it is created, the enclave can start accepting pages. The first page is assumed to be the entry point, which is why the created and building state are separate. Once

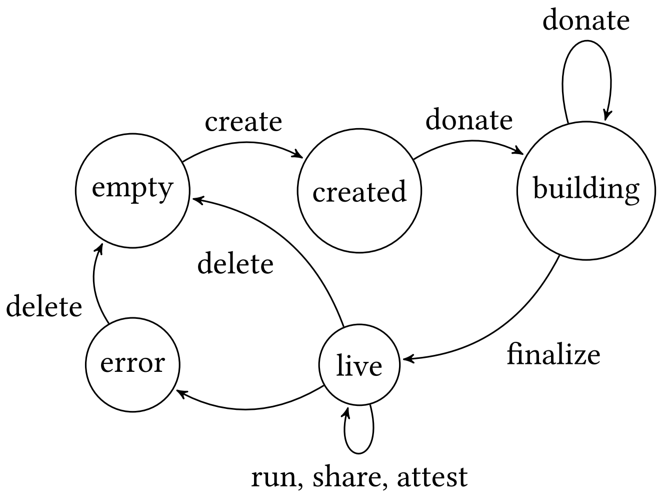

Figure 4.4: Life cycle state diagram that is tracked in enclave data entries. The transitions are labeled with the type of message that causes the transition. Any message type that is missing for a particular state is forbidden. The state transition between live and error is not caused by a message but happens when a trap occurs that cannot be recovered from.

in the building state, the enclave can accept donations of an arbitrary number of pages until the enclave is finalized. With the "finalize" step the management shim creates the attestation measurement of the content of all the enclave's initial pages. After the enclave is finalized it goes into the live state and can no longer accept any pages. In the live state the enclave can be run, can receive shared pages and can be attested. Upon deletion all enclave pages are filled with zeroes and given back to the default enclave. Once the enclave pages are given back, the enclave data entry goes to the empty state. If at any point the enclave experiences a trap that cannot be recovered from, it moves to the error state. From the error state the enclave can no longer run and is just waiting for deletion. Keeping track of these states enables the management shim to know which messages are allowed in which state of the enclave's life cycle.

4.5.3 Scheduling on enclave cores

Scheduling is managed by the "run" message specified in Table 4.1. The OS can request an enclave to be run on a specific core. The management shim has a data structure that remembers which enclave is running on which core. This data structure makes it possible for the management shim to check whether an enclave is running when deletion is requested and to perform a secure context switch between enclaves.

The complexity of deciding which enclave is run on which core is delegated to the OS and driver. This adheres to my threat model because it allows only the OS to perform a denial of service attack and it reduces the size of the TCB. In general the idea is that an enclave runs on a core until another one is scheduled there. To allow the management shim to interpose, I introduce periodic interrupts where the management shim can check whether it needs to perform a context switch or not.

The secure context switches on enclave cores are an important part of the security model. The physical isolation protects against all intra-core side-channel attacks as long as enclaves do not share a secure core. Context switching introduces time-multiplexed sharing of a core, so intra-core side-channel attacks must be prevented upon context switch on the secure cores. In essence this requires a flush of all microarchitectural state like the first-level cache, TLB, store buffer, register file, branch target buffer, in-flight instructions, etc. Previous work has shown the intricacies of purging microarchitectural state and the difficulty in identifying all the state [17, 99], so secure cores that have little state make this easier. The execution time of the secure context switch must not vary based on the core's state to avoid leaking information about enclave execution. All of the software logic for this secure context switch resides in the management shim for security reasons.

In my initial implementation of Praesidio, I choose to only run one enclave on a secure core at a time. This has the downside that the number of concurrent enclaves is limited to the number of secure cores. If more enclaves are needed at the same time then the management shim will need to securely context switch between multiple enclaves, which might cause delays. A secure core could be designed to securely multithread enclaves, but until that is done our design has the limitation that the number of secure cores limits the maximum concurrent enclaves.

4.5.4 Handling traps

Handling traps that occur within enclaves needs to be done by the management shim because traps allow for side-channel attacks if handling them is delegated to an untrusted OS. One example of such an attack is tracking enclave access patterns through page faults [35]. To avoid this the management shim installs a simple trap handler. It can do this because the management shim is the first thing that runs on all secure cores.

One example is a trap caused by a management shim interrupt in which case the management shim interposes to check for scheduling messages and either performs a context switch or resumes the enclave. The behavior of the current trap handler is to abort an enclave and put it in the error state (shown in Figure 4.4), except if the trap is the management shim's own interrupt. However, the trap handler can be extended to allow the enclave to provide its own trap handler. This trap handler can either be provided by the enclave runtime or it can use existing programming semantics for handling exceptions, such as signals.

4.5.5 Attestation

Attestation is an important part of any enclave system because it allows a remote party to verify that an enclave is running securely on a trusted platform. Existing attestation methods can be ported to a system with physically isolated enclaves. Attestation is a way for a system to "prove to a remote party that it is a valid [platform] without revealing its identity and without linkability" [100].

Table 4.1 references a "finalize" message, which can be sent to the management shim. This message causes the management shim to prevent any more pages to be donated to that enclave and to create a measurement of all the pages in the initial state. The management shim calculates this measurement using a cryptographic hash function like SHA256 or SHA3, and the measurement is stored as part of the enclave's metadata. Another important part of attestation is proving to the remote party that the enclave is running on a trusted platform. To do this Praesidio must measure and sign the current state of the management shim.

Signing can be done with any secure digital signature algorithm, which can be based on RSA (like in previous enclave systems [27]) but can also be based on elliptic curves which requires a smaller signature size for the same security level. The signature is created over a quote when the "attest" message is sent to the management shim. The quote includes the measurement of the management shim, the measurement of the enclave, "an ephemerally generated public key to be used by the challenger for communicating secrets back to the enclave" [57] and a number used only once to ensure freshness. More details of what should be included in a quote can be found in the specification of attestation in SGX [57, 101].

The message return value is sent in multiple messages because the management shim can only return a maximum of 96 bits per message and attestation requires a bigger data structure that has a quote of 3,488 bits (see `sgx_quote_t` [102]) and an ECDSA signature, which is 256 bits if the NIST P-256 parameters are used [103]. Points on an elliptic curve are usually compressed by returning just the x-coordinate because the sender can use the curve's equation to calculate the corresponding y-coordinate. Additionally another 256 bits are necessary to communicate the enclave's ephemeral public key. I will not cover the detail of all the fields in the quote, but more details on the enhanced privacy ID direct anonymous attestation scheme used by Intel SGX can be found in the paper by Brickell et al. [101].

There are also ways to do runtime attestation, where the current state of the stack and heap are taken into account [104]. Praesidio lends itself to both attestation of the initial state and during runtime.

4.5.6 Page table management

Since the management shim runs in M-mode (the most privileged mode in RISC-V), the management shim must install a page table before it can launch enclaves in less privileged mode. Currently, the management shim installs all the pages of an enclave at a fixed virtual address, so that enclaves can be compiled and use global variables within their address space without having to manage a global table for position independent code. The management shim also maps the mailboxes into the virtual memory of enclaves so that enclaves can interact with the management shim.

4.5.7 Enclave runtime

The main purpose of the enclave runtime is to facilitate communication between enclaves and to maximize code re-use. It has an API to set up communication pages and provides functionality to implement a ring buffer on the shared pages that are described in Section 4.4. The first thing most enclaves do is set up a communication channel, which is done using the following function call with `receiverID` being equal to the default enclave identifier:

```
setupCommunicationPages(receiverID)
```

Internally this sets aside one of the pages owned by the enclave to send over and sets the reader page to the enclave it wants to communicate with. Since the tag directory is not mapped into enclave virtual memory the setting of the reader is done using a environment call to the management shim. It also waits for that enclave to reciprocate, uses an environment call to map this page into its virtual memory and saves this page internally. This use of one-directional channels is similar to setting up two Unix pipes to create bidirectional communication.

Inside the enclave runtime it stores a pointer to the sending and receiving page and uses these pages in a ring-buffer fashion. The ring buffer implementation needs a status of an entry and the length of an entry. Since the ring-buffer is implemented on a page, I only need 12 bits to encode the length and 1 bit to encode the status. I combine both of these into two bytes which is equal to the length if the most significant bit is unset and if the most significant bit is set this indicates the entry is not ready yet. After these two bytes there is a payload.

The following call takes the next entry in the ring buffer, writes the payload after the first two bytes and sets the most significant bit of the next entry to 1. It then writes the length for the current entry to indicate to the receiver that it is ready:

```
writeMessage(message, length)
```

The following call waits until the most significant bit of the status is unset and then returns the corresponding message and length:

```
(message, length) ⟵ readMessage()
```

Both `writeMessage` and `readMessage` update an internal pointer for the current position in the communication pages. The implementation of a ring buffer allows new entries to start being written to the start of the page again once the end of the page is reached. Another benefit of these send and receive functions is that they automatically copy the message content from the shared page to enclave's private memory. This means that when enclaves check the content of the message there is no risk of a time-of-check to time-of-use race condition occurring.

Besides sending and receiving from a ring buffer, Praesidio also allows complete pages to be transferred by the following two commands. The first sets the reader tag for a page and communicates the address of this page via a "share" message:

```
giveReadPermission(receiverID, page)
```

On the receiving end the enclave can get the address of the shared page via the following function:

```
page ⟵ getReadOnlyPage(senderID)
```

This mechanism allows for a quick transfer of data that is larger than a page, while the ring buffer is an efficient way to send smaller messages and reusing space.

4.6 Linux driver and user API

The purpose of the Linux driver is to show how a rich OS can expose enclave functionality to its applications. The driver is primarily in charge of sending messages to the management shim about enclave life cycles and setting up a communication channel between the application and its enclave. The driver is excluded from the TCB, so it is up to the management shim to prevent any malicious requests from being processed. The purpose of the user API is to provide an abstraction layer for the application. In essence the user API provides functions to send messages and manage enclave lifetime like the ones below:

```
enclaveID ⟵ create(elfFile)
delete(enclaveID)
(quote, signature) ⟵ attest(enclaveID, challenge)
```

To create an enclave, it is important to know that the Praesidio driver consists of a base driver and an enclave driver. The user API requests an enclave device to be made for them by using an `ioctl` to the base driver, which returns the file descriptor of a new instance of the enclave driver. Having a separate device for each enclave allows the driver to limit access to the enclave based on process identifiers. The user API opens the new device file and sends an `ioctl` request for an enclave to be created using the contents of an executable and linkable format (ELF) file. After this it sets up a communication channel by requesting a page to send over and a page to receive from. The application can then call the `writeMessage` and `readMessage` calls described in Section 4.5.7. Similar to the enclave runtime, the user API also provides ways to donate complete pages using the `giveReadPermission` and `getReadOnlyPage` calls.

Internally, the Linux driver converts the `create` call into individual messages sent to the management shim. For example each page is donated individually and it requests the enclave to be finalized and run at the end of the creation process. To create the sending page the driver allocates a page, makes the enclave reader of that page and maps the page into the application's virtual memory space.

4.7 Summary

In this chapter I show how a system can physically isolate enclaves to protect them from side-channel attacks. I describe the hardware and software architecture of such an enclave system, which is used in the implementations described in Chapters 6 and 7. I show that, through physical isolation, an enclave system can exist without changing the implementation of fast application cores. Physical isolation is a powerful technique to add enclave functionality to heterogeneous multi-core systems.

Chapter 5

Security analysis of physically isolated enclaves

5.1 Introduction

In this chapter I assess the security of the Praesidio system described in Chapter 4 within the context of my threat model. The contribution of this chapter is to show that physical isolation is a feasible model to protect enclaves from side-channel and other important attacks. This contribution is essential to support the claim that Praesidio is a realistic way to protect enclaves. I also discuss Praesidio in the context of a formal model of secure remote execution, its trusted computing base (TCB) and compatibility with CHERI capabilities.

5.2 Threat model

This section summarizes the threat model presented in Section 2.8. The foundation for this threat model is protection against attacks that can be launched from privileged software, which is similar to that of previous enclave systems [13,17,27]. Enclave systems generally protect the following assets from attacks launched by privileged software and other enclaves: confidentiality of enclave memory, integrity of enclave memory and authenticity of the whole enclave environment. This means that enclave threat models include direct attacks like reading from and writing to memory, which can be launched by an operating system (OS) or a malicious enclave.

Denial of service attacks are classically excluded in enclave threat models because privileged software is in charge of resource management like memory allocation and scheduling. It is trivial for privileged software not to schedule an application or refuse to allocate memory to it. Physical attacks are also excluded because requiring physical access to a machine is less scalable than just requiring software access.

Previous enclave threat models lack consensus on to what degree to protect against side-channel attacks. Some completely ignore them, while others only consider certain side-channel attacks. This threat model includes side-channel attacks that can be launched from privileged software; like timing of memory requests and contention of execution units.

5.3 Physical isolation

I argue that it is impractical and bad for performance to gain enough confidence that intra-core side-channel attacks are covered when an attacker shares a high-performance core with a victim. I propose to protect against intra-core side-channel attacks from a policy point of view, which enforces that all enclaves are physically isolated on a separate core from other applications. Table 2.1 shows that the whole class of intra-core side-channel attacks are highlighted in gray for the "proposed" column. By construction physical isolation precludes intra-core side-channel attacks.

Another benefit of physical isolation is that it helps with the trade-off between security and performance. Application cores can use speculative and out-of-order execution to improve performance of the feature-rich OS and applications, while enclaves can run on cores that lean more towards security without having to cater to the performance requirements of other applications.

5.4 Characterizing secure remote execution

This section argues but does *not* prove that a formal specification developed at MIT [59] applies to the Praesidio enclave system [1]. Subramanyan et al. describe a trusted abstract platform (TAP), which is "a formalization of idealized enclave platforms along with a parameterized adversary" [59]. They then "present machine-checked proofs showing that SGX and Sanctum are refinements of the TAP" [59]. I argue that Praesidio is also a refinement of the TAP, which means the formalized properties also apply to Praesidio. Comparing to this formal definition of secure remote execution increases the confidence that Praesidio is trustworthy and can run enclaves securely.

5.4.1 Formal definitions

Subramanyan et al. [59] define secure remote execution as being:

> A remote platform performs secure execution of an enclave program e if any execution trace of e on the platform is contained within $[\![e]\!]$. Furthermore, the platform must guarantee that a privileged software attacker only observes a projection of the execution trace, as defined by the observation function obs [59].

The semantics $[\![e]\!]$ are defined as being the set of all allowable execution traces of an enclave, based on an initial memory content, $init_e$, and a configuration, $config_e$. To simplify their proofs, they define a subset of the total system state, σ, to be the input of the enclave, $I_e(\sigma)$. A series of inputs $\langle I_e(\sigma_0), I_e(\sigma_1), ..., I_e(\sigma_m)\rangle$ uniquely defines a single execution trace in $[\![e]\!]$, which enforces that the system is deterministic.

They formally define secure measurements, integrity and confidentiality. They also define a TAP and prove that the secure remote execution, secure measurement, integrity and confidentiality properties hold in this TAP. To accomplish this they use the BoogiePL intermediate verification language and the Z3 SMT solver [105]. In essence their formal definition of the enclave measurement, integrity and confidentiality define the security invariants of their system.

5.4.2 Applicability to physically isolated enclaves

Subramanyan et al. use a single-thread model to perform their proofs [59]. This is possible because SGX and Sanctum enclaves run on the same core as the adversary, so a single core system is possible. With physically isolated enclaves, however, the adversary and the enclaves run on separate cores. In this section I argue that the adversary running on a separate core can be modeled as performing the operations as they become visible on the secure cores. This means that single-thread enclaves running on Praesidio can be compared to those running on the TAP.

The main problem with using this single-thread model in a multi-core system is the memory model. Memory models define rules by which loads and stores have to appear in the global memory order. For example sequential consistency states that all loads and stores must respect the program order. Total store order (TSO) relaxes this model by allowing stores in a program to be observable locally before being observable globally. Another memory model is RISC-V weak memory ordering (RVWMO), which is defined in Chapter 14 of the instruction set manual [6] and specifies thirteen cases under which program order must be respected.

This plays into my model because part of the TAP is defining all the operations that can lead to state transitions in a system. Examples of operations that interact with memory are `fetch`, `load` and `store`. If these fetches, loads and stores interact with the global memory, then it becomes important whether the memory model is TSO, RVWMO or something else. A system with multi-thread enclaves would have to take this into account. However, if enclaves consist of only a single-thread, then a system designer can model the attacker not as the operations they perform on global memory, but on the memory that is visible locally to the enclave. This means that even though the enclave operates in a multi-core system, the proof only needs to consider the memory locally visible to each enclave.

Additionally, single-thread enclaves do not interact with the memory model in any complicated way because the memory it handles is solely owned by that enclave. No other application can read or write the enclave's memory. The only exception to this rule is shared pages and in my system these are only writable by the owner of the page. This means that even for shared memory, only one enclave can store to that memory and the reader can only load from it. The results of stores to shared memory become visible to a secure core in a certain order and so the attacker's `store` operations are modeled only at the point where they become visible to the secure core. In this reasoning each secure core is modeled separately because different secure cores may perceive memory operations in different orders.

The benefit of this reasoning is that I can leave the BoogiePL verification model unaltered and the proof that the TAP fulfills the security requirements still holds. If Praesidio supported multi-thread enclaves, this assumption must be revisited, and the memory model would need to be considered as well as whether the cache coherency protocol implements the memory model correctly.

5.4.3 Review of Praesidio's memory protection

Praesidio keeps track of which page is owned by which enclave, and this is described in more detail in Section 4.4. Each physical page is tagged with the corresponding enclave identifier. Besides the owner tag, Praesidio also allows a reader to be added to a page, which enables for one-directional communication. All of these tags are stored in a tag directory and the memory protection provided by them is orthogonal to the protection provided by memory translation. This is deliberately done so that privileged code can still hold complete control over its own memory translation and which memory is reserved for enclaves, while simultaneously ensuring the integrity and confidentiality of enclave memory. The page tables of the secure cores are installed by the management shim. In my system first-level caches of secure cores are private and, due to physical isolation, only hold data from the currently running enclave. The last-level cache (LLC) keeps track of the tags for each cache line, which allows the cache to be shared while still enforcing access control. The LLC is also partitioned to prevent side channels through this cache.

5.4.4 Refinement of the trusted abstract platform

To show that Praesidio is a refinement of the TAP [59], I must show that for all of the state variables and allowable operations there are equivalent variables and operations in Praesidio. The following list shows the state variables of the TAP with a description of what they are and how they map onto Praesidio:

- `pc`: The program counter is equivalent to my program counter.
- `regs`: The architectural registers are equivalent to my RISC-V register file.
- `mem`: The memory is equivalent to my dynamic random-access memory (DRAM).
- `addrmap`: The mapping of virtual addresses to physical addresses as well as permissions for the current process. Subramanyan et al. [59] use this variable to simulate memory translation as well as memory protection. It directly maps onto the page table in the processor, since the security monitor controls this on all cores. My system works slightly differently because protection is based on physical pages and controlled by `owner` and `reader` tags. I claim that my protection is equivalent to this method even though in hardware the enforcement happens by the tag directory as opposed to the page table. Subramanyan's system enforces access control upon installation of the page table in each core, while my system enforces access control after translation inside the memory hierarchy independent of each core. Even though our systems are different, my platform still maps onto the TAP. This is because I define `addrmap` to be equivalent to the page table except in the case where the current enclave does not own the piece of memory. For those pages, all permissions are unset unless the enclave is the reader of that page in which case only the read permission is enabled.
- `cache`: The cache is equivalent to my LLC.
- `currentenclave`: The current enclave is equivalent to my current enclave identifier control and status register (CSR).
- `owner`: The map from physical addresses to the enclave it is allocated to. In my system the tag directory keeps track of the owner of each page, which is the same as in the TAP. I extend this variable with a reader tag, which is used to enforce changes to the `addrmap` variable.
- `encmetadata`: The map from enclave identifiers to a metadata record type, which is equivalent to my enclave data records.

- `osmetadata`: In their TAP this variable is actually the metadata of the OS. Due to the physical isolation, I must still have this variable, but it has a slightly different meaning. The steps an attacker takes and this variable are actually projections of the attackers actions as they become visible to the enclave core. As I argue earlier in this section, the stores and loads the attacker performs must be mapped from the core that they actually occur on to when they become visible to the enclave. This avoids having to consider the memory model in this proof, but still guarantees the security properties. Thus this `osmetadata` is actually a projected variable that performs the attackers TAP operations as they become locally visible to the enclave core.

Part of the state variables is the enclave metadata, which is composed of a number of fields. In the following list I present how each field is present in the enclave data records used in Praesidio:

- `entrypoint`: The enclave entry point in Praesidio is the start of the first page donated to the enclave.
- `addrmap`: The address map is a mapping of virtual to physical addresses and its permissions. In Praesidio each enclave has a page table and due to how the memory hierarchy works, permissions for pages that the enclave own are allowed to be set to any value, for pages that the enclave is a reader only the read permission is set and otherwise all permissions are unset.
- `exclvaddr`: These are the virtual addresses that are exclusively accessible to the enclave. This is equivalent to the pages that are owned by the enclave and lack a reader tag.
- `measurement`: The enclave measurement is created once “finalize” is called on an enclave in Praesidio.
- `pc`: This saved program counter is used when trapping to the management shim.
- `regs`: The saved register state is used when trapping to the management shim.
- `paused`: Praesidio keeps track of all active enclaves in the system (maximum of one per secure core). If an enclave is not marked as running on any core, this is equivalent to the `paused` variable being set.

The final part of showing equivalence of Praesidio to the TAP is the mapping of the TAP operations:

- `fetch`, `load`, `store`: Fetch, read or write from/to memory. For showing equivalence with Praesidio, memory is considered to be locally visible memory as I discuss in Section 5.4.2.

These operations must fail if they are not executable, readable or writable respectively according to the `addrmap`.

- `getaddrmap`, `setaddrmap`: This gets or sets the `addrmap` variable, which is only done by the management shim in Praesidio. Praesidio allows all changes to the `addrmap` as long as the current enclave owns that piece of memory. It also allows entries to be added with just the read permission if that enclave is a reader on that piece of memory.

- `launch`: The launch operation initializes an enclave by allocating an entry for it in the `encmetadata`. In Praesidio, the management shim creates an enclave data record and to achieve a full launch, the management shim must receive a "create" message, several "donate" messages and a "finalize" message (see Section 4.4).

- `destroy`: Sets the enclave memory to zero and removes it from the `encmetadata`. In Praesidio this occurs when the management shim receives the "delete" message.

- `enter`, `resume`: In my system enclaves run in parallel on physically isolated secure cores, so entering and resuming an enclave is nonsense. Instead the OS can tell the management shim to schedule an enclave on a core by using the "run" message.

- `exit`, `pause`: Exiting is a system call by the enclave to indicate it is done, which frees up the secure core for other enclaves to be run. Pausing saves a checkpoint of the program counter and the registers into the `encmetadata` entry.

- `attest`: Returns a hardware-signed message of a nonce and an enclave measurement. More details on how attestation is done in Praesidio can be found in Section 4.5.5. In Praesidio this message is generated by the management shim upon receiving an "attest" message.

Subramanyan et al. also model one cache side-channel attack and they claim that as long as addresses from different enclaves are never mapped to the cache set then this cache timing attack is impossible [59]. There are two limitations with this approach: it ignores the numerous other side channels that exist in the memory hierarchy (e.g. cache request buffer contention and DRAM row buffer collisions) and it disallows approaches like random replacement policies which significantly reduce the bandwidth of this side channel.

To be sure that my model is a refinement of the TAP, I would need to extend the BoogiePL verification model and rerun the SMT proofs. Unfortunately, this is a significant piece of work and thus outside the scope of this PhD. In this section I argue qualitatively about how my system is a refinement of the TAP by showing how each state variable and operation is implemented in Praesidio. This gains confidence in that there exists a formally defined model for Praesidio that can be proven equivalent to the TAP in the same way that Subramanyan et al. show that Intel SGX and Sanctum are equivalent [59]. This also gains confidence that the security properties of secure remote execution are upheld in Praesidio.

5.5 Trusted computing base

As part of the threat model, Praesidio explicitly trusts certain parts of the system. The following list of items are included in the TCB:

- The tag directory, the LLC and the translation of tags between the two.
- The memory interfaces between the fast cores and the LLC.
- The implementation of the secure cores.
- The management shim and securely booting into the boot memory.
- The key distribution system and certification scheme used to verify the attestation keys in the system.

These parts must be trusted to ensure that access control is enforced, to guarantee that the tags are preserved across the memory hierarchy and to ensure that tags are only modified in legal ways. The management shim is also trusted to ensure that enclaves can securely bootstrap with the system and prove this through remote attestation. To do this there is a secret platform key that can only be accessed by the management shim. The secrecy of this key is ensured by the tagging system and by the boot process guaranteeing that the management shim is the first code to run on the system. Additionally, the hardware implementation of the secure cores are trusted to correctly execute management shim code, keep the attestation keys secret and provide functionality to securely context switch between enclaves.

5.5.1 Exclusions

In accordance with the threat model, there are also parts of the system that are explicitly excluded from the TCB to protect against the defined attacker model:

- Fast cores
- Privileged software running on fast cores (including OSs and hypervisors)
- Linux driver
- User applications and user application programming interface (API)

- Other enclaves and their runtime environments

Excluding all of these from the TCB gives engineers the freedom to pursue performance optimizations on fast cores and to create rich features in OSs without having to consider the enclave threat model.

It is also important that the Linux driver is untrusted because this is the piece of code that handles complex tasks like memory management and scheduling of enclaves. The management shim contains the logic to check whether the actions of the driver are authorized or not, but it lacks the need to implement all the logic to do the tasks itself. This is essential to decrease the size of the TCB, which is often used as a measure for how easy it is to gain confidence in the security of a system.

5.5.2 Code size

The management shim, as presented in Section 6.5.3, consists of 545 lines of code [106]. This is less than the 1,600 lines of code that make up Lee's security monitor [58]. It is also significantly less than the 8,700 lines of code in seL4, which is a formally verified OS kernel [11]. The relatively small size of the code base means that if formal verification were a requirement, it would be feasible to rewrite the management shim in the subset of C used by the seL4 kernel. Using a subset of C supported by a formally verified compiler written in higher-order logic means the shim's functionality can be formally expressed. Researchers can then write a formal specification of the management shim's functionality in the same higher-order logic and use a theorem prover to prove both to be equivalent. It is unlikely that this process would expand the size of the management shim's codebase by more than one order of magnitude. Initially it took seL4 more than 2 person years to create their first implementation, excluding the specification and proofs [11]. So although it is feasible to do the same for the management shim, it would take at least a few person months to produce this and therefore it was left out of scope for this thesis. Having a practical pathway to formal verification is crucial to gain more confidence in the TCB.

5.6 Capabilities

Another way to gain more confidence in the security critical software in my system is to use CHERI capabilities [107]. CHERI introduces hardware-enforced bounded pointers with permissions. CHERI protects against many common attack vectors such as buffer overflows, return oriented programming and buffer over-read attacks, which are often an essential part of a complete exploit. Microsoft security response center estimates that 67% of the exploits reported by them in 2019 would have been mitigated by CHERI [108]. The way to use CHERI capabilities in security critical code is exemplified in CheriOS [109], which is a kernel that allows compartmentalization of every part of the OS. Having the secure cores be CHERI-enabled is an effective way to make both the management shim and enclaves more robust against common software attacks.

5.7 Attacks

This section walks through the attacks that are included in my threat model (Section 2.8) to describe how Praesidio protects against them.

5.7.1 Direct accesses

Directly reading from or writing to protected pages are examples of attacks using direct accesses. Praesidio protects against these attacks by tagging physical pages and enforcing access control using the LLC and the tag directory (see Section 4.4). Additionally, the management shim fills pages with zeroes when deleting enclaves, so there is no way for an attacker to reclaim a page and then read sensitive content.

Another way of performing the same attack is by accessing protected memory through another memory device using direct memory access (DMA) requests. DMA requests in my system are currently only acting on behalf of the fast application cores. For example an OS can write a shader for a graphical processing unit (GPU) to access enclave memory. To solve this problem each memory device must adhere to the tags in the tag directory and an input-output memory management unit (IOMMU) must enforce tags for peripheral devices.

5.7.2 Code alteration

Since the OS supplies the pages that contain the enclave code, there is nothing stopping the OS from altering the code before donating the pages to an enclave. Just like other enclave systems, my system makes these code alterations detectable by a remote party. This process is called attestation and my system creates a hash of the initial state of the enclave during the "finalize" step, which is used in creating an attestation signature for remote parties to verify. If enclaves alter their own code, this leaves the measurement unchanged because the "finalize" step happens before the first run of the enclave. This attestation signature is generated using a platform key, which needs to be hard-coded into the hardware and protected from leaking outside of the management shim. More details on the attestation process are in Section 4.5.5.

Remote attestation allows a remote party to verify an enclave before starting to communicate. For example a client can spin up an enclave in the cloud and verify that it is set up securely before sending sensitive data to it. Another example is a user authentication enclave that can be verified by a remote web service before commencing with the user authentication process, which is a process standardized by FIDO [48].

5.7.3 Side channels

Table 2.1 is split between intra-core side-channel attacks that require the victim to be running on the same core as the attacker and inter-core side-channel attacks that lack this requirement. All of the side-channel attacks can be launched by privileged software, usually using timing to measure contention of shared resources.

Intra-core side-channel attacks rely on contention of resources that are local to a core. Sharing a translation look-aside buffer (TLB), for example, allows for contention that leaks memory access patterns [35]. Execution units [37] and hardware accelerators [65] leak information on what operations other threads are doing, while the branch predictor leaks information on control flow [39]. Intra-core side-channel attacks are included in my threat model and are the main motivation to explore physical isolation as a defense mechanism.

Physical isolation provides protection against intra-core side-channel attacks by making sure that core resources are not shared between enclaves and attackers. Even though enclaves do not simultaneous share resources, there is still the opportunity for time-multiplexed sharing of resources because of context switching between different enclaves on secure cores. It is imperative that context switches scrub all the local microarchitectural state or partition it securely (see Section 4.5.3).

Inter-core side-channel attacks rely on contention on resources that are shared by different cores like caches and DRAM. In caches, cache lines and cache request buffers are shared which leads to a timing dependence between cores [27]. Similarly in DRAM row buffers containing the currently open row [30] and request buffers containing a queue of DRAM requests [17] cause timing dependence between cores. All of these attacks are included in the Praesidio threat model and requires a careful design of the memory hierarchy to ensure isolation.

Since the memory hierarchy is the main resource that is shared between attackers and victims in my system, contention in the memory hierarchy is an important aspect to solve. Contention happens when two applications compete for the same resource, which creates information leakage because it creates a time dependency between these two attackers. As an example, enclave systems have mitigated cache line contention by using static partitioning schemes [17,27], and flexible partitioning schemes provide an alternative approach with lower performance overhead for applications that heavily use the LLC [89,97]. Physical isolation can be paired with these cache mitigations as well as with other memory contention mitigations like partitioning memory request buffers [17].

Trap side channels

These attacks are impossible because the management shim catches traps like page faults on the secure cores. This way the attacker running on the fast cores or on other secure cores cannot attain this information.

Execution time

Sensitive data can leak through variations in execution time. Due to my system physically isolating enclaves onto secure cores, many side-channel attacks that affect execution time are solved and enclave developers no longer need to worry about such attacks. However, if enclaves make their control flow dependent on secret information, simply the time between enclave responses can leak information. This is defined to be outside of the scope of my system, so it is the enclave developer's responsibility to ensure their code does not leak sensitive data this way. It should be relatively intuitive for developers to avoid data-dependent control flow, especially when compared to the care needed to avoid microarchitectural side channels.

DRAM bit leakage

DRAM bit leakage can be performed by repeatedly accessing adjacent rows [30] and is within my threat model. It can be used in an active eavesdropping attack to read confidential information like enclave memory [64]. Recent attacks have shown that the current row refresh techniques implemented by DRAM vendors are insufficient to protect against these attacks [110]. Physical isolation of enclaves as a technique is orthogonal to the mitigations of DRAM bit leakage, which allows for seamless integration into future enclave systems as DRAM vendors find adequate solutions to this vulnerability.

Cache and DRAM collisions

In Chapter 3, I analyze side channels that depend on cache collision and DRAM collision. My system can be configured to use a static or flexible partitioning scheme in the LLC to avoid the cache collision problem. The other side-channel attacks mentioned in Section 3.4 are solved by carefully designing the memory hierarchy [17]. In Chapter 6, I present an implementation of Praesidio on Spike, an instruction-level simulator for RISC-V. The methodology to transform side channels into direct channels, described in Chapter 3, can then be used to analyze the security of this implementation. For example the methodology can verify whether the different LLC configurations are secure against "prime and probe" attacks.

Flush and reload

One possible configuration of my LLC involves a random replacement policy. This stops the attacker from being able to find a colliding address with the victim. However, this does not stop a "flush and reload" attack. To show that this attack is possible consider two collaborating processes that use this channel for transmission:

1. Receiver fills most of the cache with its own data.
2. Depending on the secret bit, the transmitter does nothing or fills most of the cache with its own data.
3. Receiver checks some of the last entries that it used in step 1. If most of them are still there then the secret bit is a 0, otherwise the secret bit is a 1.
4. Go to step 1 until done transmitting.

This attack needs some type of error correction code, since the random replacement policy causes the steps of the attack to have a probabilistic effectiveness. This attack can transmit 1 bit per iteration, but requires filling the LLC in step 1 and 2. This means that the time taken to transmit N bits is on the order of $O(N \times S)$, where S is the size of the LLC.

This attack shows a disadvantage of using random replacement over static partitioning. However, it is important to note that this "flush and reload" attack is less powerful than the "prime and probe" attack. The bandwidth is lower because of the time necessary to flush the complete LLC. In the "prime and probe" attack, the attacker is able to detect whether a small subset of the cache has been accessed. With the "flush and reload" attack, the attacker can only tell whether there has been an access to the whole cache. This means that any other core running in parallel creates noise in the attack. Another main downside of "flush and reload" is that flushing the cache is an expensive operation that can more easily be detected by an intrusion prevention system.

One caveat to this argument is that optimization to this attack might exist, like using probabilistic eviction sets [111]. These optimizations might make the "flush and reload" attack more powerful and may require the use of the static partitioned cached as opposed to the flexibly partitioned one.

5.7.4 Transient execution

Transient execution attacks are a special class of attacks that exploit the information leakage caused by processors speculatively executing instructions [16]. Protection against transient execution attacks follows the same reasoning as that of intra-core side-channel attacks. Transient execution attacks cannot succeed because attackers do not share the same core as an enclave, and on an enclave context switch all speculative state is purged or securely partitioned [17, 60, 99].

5.8 Summary

In this chapter I describe how Praesidio compares to a formal specification of secure remote execution, discuss the TCB and show how Praesidio protects against the attacks included in my threat model. Table 2.1 shows that physical isolation prevents side-channel attacks through the TLB, execution units, branch predictor, transient execution and hardware accelerators, simply because physically separate cores do not share these units with any other core. Inter-core side-channel attacks become more challenging as a result of the changes made to the memory hierarchy. These changes are orthogonal to other mitigation techniques. Section 5.7.3 discusses how my proposed system prevents side channels through collisions in cache lines, cache requests, DRAM rows, DRAM requests and DRAM bit leakage. Each of these aspects increase the confidence in Praesidio and the approach of physically isolating enclaves onto secure cores protects enclaves from the threats against them.

Chapter 6

Evaluating Praesidio in simulation

Note: Most of this chapter is based on a paper published by the author of this thesis [1].

6.1 Introduction

As Praesidio is a new system that physically isolates enclaves from applications, I need to evaluate whether it is a valid approach with regard to performance and overhead. Chapter 4 contributes to the field by describing a system that combines the security of physical isolation with the flexibility of enclave trusted execution environments (TEEs). This chapter builds on that contribution by evaluating the communication, creation, area and memory overheads of the design to show that not only is it a theoretical way of creating an enclave system, but it also practically implementable with reasonable costs. To do so this chapter presents a Linux driver, a trusted management shim and a hardware simulator, which together embody the first implementation of Praesidio.

I implement Praesidio on Spike, a RISC-V instruction set simulator. Spike includes a cache simulator, which I configure with the parameters shown in Table 6.1 and are the same as Berkeley's out-of-order RISC-V processor (see Figure 1 in the introduction of the BOOM documentation [2]). The data and instruction cache sizes align well with those from Intel, but Praesidio's last-level cache (LLC) is twice the size of Intel's second-level and a sixteenth of the size of the third-level cache [3].

Besides the cache configuration, I add a tag directory to Spike and enforce access control on each memory access. For all of the experiments I change the interleave to 6, which means that each core takes a turn executing six instructions before the next one. The reason for using six is that any lower than that Linux no longer boots reliably. The closer the interleave value is to one the more realistic the simulation is compared to simultaneously executing instructions.

Cache	Sets	Ways	Line size (B)	Total size (KiB)
Data	64	8	64	32
Instruction	64	8	64	32
Last-level	1024	8	64	512

Table 6.1: Parameters of the cache hierarchy, which are based on the BOOM architecture [2]. The line size is the same as Intel's Core i7 processors (see Table 2-31 in the architecture reference [3]).

All of the experiments run on Linux 4.15.0 using buildroot release 2016.05 and RISC-V GNU tool-chain v20171107. I have made all of this open source [112].

6.2 Enclave communication

Enclave communication is the most common interaction with enclaves. I evaluate this communication in two ways: sending messages that are smaller than a page using a ring buffer and by donating complete pages to enclaves.

6.2.1 Ring buffer

To show how the communication via ring buffer over shared memory performs between enclaves, I send messages of varying sizes. Figure 6.1 shows the instruction count and LLC accesses for varying packet sizes up to just below a page. For sending data that is larger than a page Praesidio provides a way to donate complete pages. The instruction count increases linearly with packet size, which is expected since the instruction count is dominated by the loops that write to and read from shared memory. Cache accesses are also linear because they are dominated by the loads and stores to shared memory. Receiving dominates the cache accesses because a read causes the dirty line from the writer's cache to be sent to the LLC and causes an access by the reader. In essence a write does not immediately incur the cache access penalty, while the read causes both a write-back and a read access.

6.2.2 Page donation

Besides using a single page to send messages through a ring buffer, Praesidio allows sending complete pages to enclaves. To measure the performance of this, I create a benchmark in which a user application fills a page with data, sends that page and waits for acknowledgment from the enclave. The enclave waits for the page to be sent, then reads the content of the page and sends an acknowledgment to the user application. This micro-benchmark takes a median

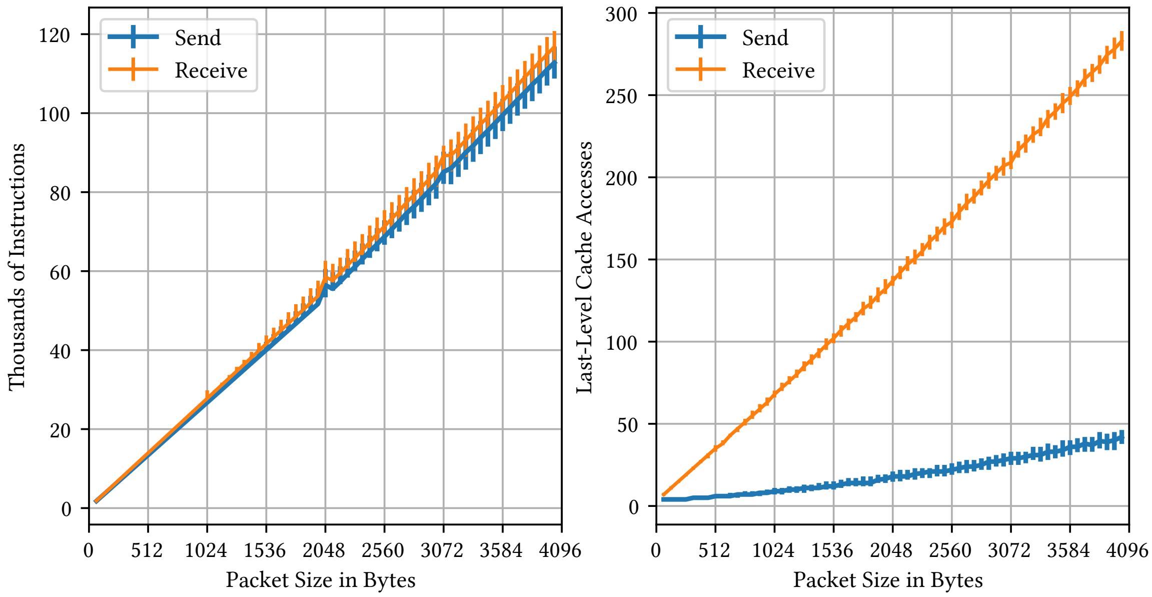

Figure 6.1: Ring buffer performance over shared pages between enclaves. Each packet size is sent 256 times, and the graph shows a line of the median value and error bars from the first quartile to the third quartile.

of 14,500 ± 500 instructions and 390 ± 20 LLC accesses, where the spread is determined by the first and third quartile using the following formula: Max(median − Q_1, Q_3 − median).

This benchmark uses significantly fewer instructions than the ring buffer benchmark because the ring buffer writes one byte at a time, while this page benchmark writes in eight byte chunks. Multiplying the instructions to donate a page by 8 gives 116,000, which is close to the trend shown in Figure 6.1.

This benchmark can also be compared to doing a similar benchmark using Unix Pipes, which is a commonly used form of unidirectional communication. Sending 4 KiB over a Unix pipe on Praesidio takes 71,055 ± 3 instructions and 280 ± 30 cache accesses. Unix pipes take nearly 5 times as many instructions due to the overhead of the operating system (OS), but the cache accesses are of the same order of magnitude as my benchmark. The cache accesses are similar to my page benchmark because Praesidio enforces that enclaves cannot share first-level caches, while this requirement does not apply to two Unix processes communicating through pipes. Sending individual pages can be repeated for larger pieces of memory that need to be sent to enclaves.

	Instructions	Cache accesses
Prepare memory	2.7%	4.1%
Driver setup	96.9%	92.9%
Enclave setup	0.4%	3.0%
Total (thousands)	9,359 ± 6	79.8 ± 0.1

Table 6.2: Setup cost for creating enclaves with proportion of the different phases of the process and the total cost.

6.3 Enclave creation

Creating enclaves is a one-time cost per enclave. The creation of enclaves is done as described in Section 4.6 and can be split into three stages: preparing the enclave memory, setting up the driver and setting up the enclave. Firstly, preparing the enclave memory involves opening an executable and linkable format (ELF) file and reading the contents into user-owned pages. Secondly, setting up the driver involves creating a dedicated character device for each enclave. Finally, setting up the enclave itself involves copying the content of user pages to kernel pages that allow direct memory access (DMA), sending messages to the management shim and setting up the communication channel. Table 6.2 shows the number of instructions as well as the number of LLC cache accesses in the different phases of creating an enclave. The driver setup takes the most instructions and cache accesses. This means that I can gain only marginal improvements in the creation process from optimizing the management shim and user application programming interface (API).

6.4 Hardware area and performance

The main added hardware costs are the added hardware area required for the extra cores and the additional memory required. To put these costs into perspective, I examine previous enclave systems and how much additional hardware they usually add to each core. I then compare these costs with adding relatively simple cores. These simple cores are a placeholder for the secure cores, which I envisage to be small in-order cores with minimal microarchitectural state so that securely context switching between isolation domains is easier. Table 6.3 summarizes my findings where the number of gates for previous work is compared to adding a simple core. The additional hardware needed to allow enclaves to run on an existing core is similar to adding a small core that is dedicated to enclaves.

The gate comparisons made in Table 6.3 are based on research hardware or non-application cores, so to get a better idea of what the area costs would be in a real system, I look at Apple's A12 system on chip (SoC), which implements a big-little scheme where bigger high-

Solution name	Number of thousand gates
AEGIS [34]	205.0
Bastion [56]	30.1
Iso-X [43]	1.0
Sanctum [27]	5.6
Cortex M0 [113]	12.0
Twine [114]	9.4

Table 6.3: Comparison of enclave hardware cost with “little” cores.

Block	Area (mm^2)	Percentage of total
Total die	83.27	100.0%
CPU complex	11.90	14.3%
Big core	2.07	2.5%
Little core	0.43	0.5%

Table 6.4: Area of different cores in Apple’s A12 SoC using TSMC N7 process node [4].

performance cores are paired with smaller power-efficient cores [4]. The area of the two types of cores is compared to the total die area in Table 6.4. The central processing unit (CPU) complex includes 2 big cores, 4 little cores and all the hardware shared between cores. It is noticeable that the CPU complex takes only a fraction of the complete die and that the little cores are about a fifth of the size of a big core. This means that adding little cores is relatively cheap in terms of SoC area.

Assuming that the secure cores in Praesidio are similar in size to the little cores and the fast cores are the big cores, I can estimate how much it would cost to add physical isolation to an SoC like Apple’s A12. If a system has 4 little cores for enclaves, then the hardware area would increase by $4 \times 0.43 = 1.72\text{mm}^2$. This would increase the CPU complex by $1.72/11.90 \approx 14.5\%$ and the total SoC size by $1.72/83.27 \approx 2.1\%$. This seems acceptable in the era of dark silicon, where most of the die is turned off due to power constraints [93]. Additionally, this measured overhead is likely to be an overestimate because using even smaller cores for enclaves is probably better for security.

Frumusanu [4] also compares the performance difference between the big and little cores of Apple’s A12, which are summarized in Table 6.5. In essence it shows that for an approximate 4 times slowdown an enclave can experience increased security. Additionally, offloading tasks to smaller cores has the benefit of less energy consumption for the same workload and freeing up the bigger cores for other tasks. A hardware implementation of Praesidio is presented in Chapter 7.

Core Type	Performance (SPECSpeed)	Energy (kJ)
Big	44.92	9.521
Little	12.12	3.968
Ratio (Big/Little)	3.71	2.40

Table 6.5: SPECInt2006 performance comparison between the big and little cores of Apple's A12 SoC [4].

6.5 Memory overhead

I expect that the area and performance overhead due to logic to be acceptable, but there is also an overhead incurred by needing additional memory. Memory overhead is caused by the additional tags in the tag directory and the LLC as well as the storage needed for the mailboxes and the management shim.

To calculate the memory overhead, I first determine the size of the enclave identifier. If a system needs a unique identifier for the maximum number of enclaves that can concurrently exist, that is the same as how many pages can exist on the system. RISC-V allows a maximum of 56-bit physical addresses [115] and pages that are equal to or greater than 4 KiB. This means that to uniquely address each page in a system it needs at most $56 - 12 = 44$ bits. This is an upper bound because it is dependent on how much physical memory there is on a system and the size of pages.

Another way to estimate the size of an enclave identifier is by how many processes a modern OS can manage concurrently. On Ubuntu 18.04 the maximum process identifier[1] is $32{,}768 = 2^{15}$ for 8 CPUs, so each running process can be uniquely identified with a 15 bit identifier. However, process identifiers can be reused so uniquely identifying all processes that have ever run may require more than that.

In Praesidio I choose an enclave identifier of 32 bits (4 bytes). This identifier fits nicely in between the overestimate of 44 bits and the underestimate of 15 bits.

6.5.1 Tag directory

Next I estimate the size of the tag directory, which needs to be stored in dynamic random-access memory (DRAM). For each page in the system, there are potentially two enclave identifiers: one for the owner and one for the reader. The worst-case size of the tag directory is thus $2 \times \text{idSize} \times \text{numPages}$, I compare this to the size of DRAM which is $\text{pageSize} \times \text{numPages}$.

[1] The maximum value of the process identifier can be found on Linux by running the following command: `cat /proc/sys/kernel/pid_max`
This number does increase when a large number of CPUs are available on the system, which can be found using: `lscpu`

The DRAM size would thus increase by $\frac{2 \times \text{idSize}}{\text{pageSize}}$ (after removing numPages from the numerator and denominator). Assuming that pages are 4 KiB means that DRAM increases by $\frac{2 \times 4\text{ B}}{4\text{ KiB}} = \frac{2 \times 2^2}{2^2 \times 2^{10}} = 2^{-9} \approx 0.2\%$. This is a worst case estimate, since a system designer can limit the number of pages that can be tagged.

6.5.2 Last-level cache

The memory overhead in the LLC is more critical than in DRAM, this is because the size of tag compared to a cache line size is more significant than compared to a page size. In Praesidio a cache line contains 512 bits (64 Bytes) of data [3], a valid bit, a dirty bit and tag bits. RISC-V allows 56 bit physical addresses, but to avoid underestimating the overhead of the enclave tags, I assume that we have 30 bits of addressing, which is enough to address a DRAM of 1 GiB in size. The tag size would then be addressSize $- \log_2(\text{dataBytesPerLine}) - \log_2\left(\frac{\mathit{cacheSize}}{\text{dataBytesPerLine}\times\text{numWays}}\right) = 30 - \log_2 64 - \log_2\left(\frac{512\times 1{,}024}{64\times 8}\right) = 30 - 6 - \log_2 1{,}024 = 14$ bits. The size of a cache line is $512+1+1+14 = 528$. The overhead for the LLC is $\frac{2 \times \text{idSize}}{\text{cacheLineSize}} = \frac{2 \times 32}{528} \approx 12\%$. This is quite significant, but a system designer can decrease this memory cost if they are willing to add extra administrative tasks to the management shim. For example a designer can map the 32 bit enclave identifier to a smaller LLC tag, which only has to differentiate between all running enclaves. This tag can be as small as the number of cores in the system, even if the system contains 32 or 64 cores, this would reduce the tag bits to $2 \times \log_2 32 = 2 \times 5$ and $2 \times \log_2 64 = 2 \times 6$ and the overhead to about 2%. However, this would also mean that on each enclave context switch the management shim must adjust the tag mapping and invalidate LLC lines.

To minimize this cost, I figure out how many enclaves are likely to be running simultaneously, which is unlikely to be more than the number of processes allowed in the rich OS. Again, on Ubuntu 18.04 the maximum process identifier is $32{,}768 = 2^{15}$. Tags of 15 bits are enough to uniquely represent these processes, which decreases the LLC cost to $\frac{2 \times \text{smallIdSize}}{\text{cacheLineSize}} = \frac{2 \times 15}{528} \approx 6\%$. This requires the management shim to keep track of which LLC tags are mapped to which full enclave identifiers. Costs in the LLC can range from 2% to 12%, and 6% seems like a good middle ground that balances both LLC usage and the frequency of needing to change the tag mappings.

Besides the tag overhead in the LLC, I calculate the overhead caused by the mailboxes. Since each core has just one slot, the memory overhead is $\frac{\text{coreNumber} \times \text{messageSize}}{\text{cacheSize}}$. The message size can be derived from Table 4.1 by taking the biggest message payload and adding a sender and receiver identifier: messageSize = typeSize + idSize + addressSize + 2 × idSize $= 8 + 32 + 64 + 2 \times 32 = 168$ bits. Assuming the number of cores is limited to 64, this would make the overhead $64 \times 168 = 10{,}752$ bits. Compared to the LLC size of 512 KiB $= 512 \times 8 \times 2^{10}$ bits $= 4{,}194{,}304$ bits, the LLC overhead for this messaging system is $\frac{10,752}{4,194,304} \approx 0.3\%$. This calculation is a worst case because it does not account for the smaller tags used in the LLC and even then the calculated overhead is an order of magnitude less than the overhead of the tags. Thus, the total memory overhead in the LLC is dominated by the tags.

6.5.3 Management shim

Another source of memory overhead is the size of the management shim. At the moment I put the complete management shim in trusted boot memory, although this can be split into multiple parts using a secure boot process. The management shim is 2.1 KiB and 545 lines of code [106]. This is comparable the 1,600 lines of code of Lee's security monitor [58], which has similar functionality. Assuming that binary size is directly proportional to lines of code, this would make the boot memory between 2 and 6 KiB.

6.6 Discussion

The goal of this chapter is to show that the performance is acceptable for the Praesidio design. Specifically the focus is on lightweight compartments of an application to be isolated in an enclave, as I describe in Section 2.6. In case hardware vendors believe in providing a heterogeneous multi-core system to hit multiple points in the performance-security space, the lightweight compartment use cases provide compelling reasons for developers to choose a secure core even though performance would be better on fast cores. This chapter shows that the performance of physically isolating enclaves onto separate cores does not decrease communication performance significantly, which is likely to be enough based on the lightweight enclaves described in Section 2.6. In turn these use cases are also good reasons for hardware designers to make the extra area and memory available to support this system design.

6.7 Summary

In this chapter I examine the performance, memory and hardware implications of Praesidio. My evaluation shows acceptable performance cost in the following ways: communicating with enclaves causes a similar number of cache accesses to communicating over Unix pipes, adding secure cores to a modern SoC would increase the hardware area by less than 2% and storing the extra tag data increases the size of the LLC by 6% and DRAM by 0.2%. Having evaluated Praesidio using simulation, I look at a hardware implementation in Chapter 7 to gain further understanding of the performance implications.

Chapter 7

Evaluating Praesidio in hardware

7.1 Introduction

Separating fast cores and secure cores is essential to Praesidio because fast cores can implement performance features including out-of-order and speculative execution without adhering to stringent security requirements. These performance features create a significant amount of microarchitectural state, which opens the door for side-channel attacks on enclaves. The secure cores on the other hand are designed with the idea that enclaves should be protected against all software-based side-channel attacks. In essence fast cores are designed with the idea that the operating system (OS) is trusted, while secure cores are designed under the assumption that any other piece of software outside the enclave is hostile.

Praesidio isolates enclaves onto separate cores that are slower, but more secure. Paying this performance penalty is acceptable for the lightweight compartments described in Section 2.6. However, the protection mechanisms in the memory hierarchy may only insignificantly deteriorate the performance of the fast cores.

In Chapter 6 I discuss an implementation on Spike, a RISC-V instruction-level simulator. This implementation also includes a Linux driver, a tag directory and a management shim. The tag directory has a mapping from physical pages to enclave identifiers, and it keeps track of each page's owner and reader. This simulator is good for showing how software might run in an enclave and to verify that it is practical for developers to use, but it is too removed from a physical implementation to analyze the properties of Praesidio in hardware. Additional access control logic within the memory hierarchy is likely to have an impact on the memory access latency, which cannot be measured using an instruction-level simulator.

In this chapter I study the increase in average memory access time and hardware area usage by creating an implementation of Praesidio for field-programmable gate arrays (FPGAs). This chapter contributes to the field by showing that curtailing the access of fast cores to

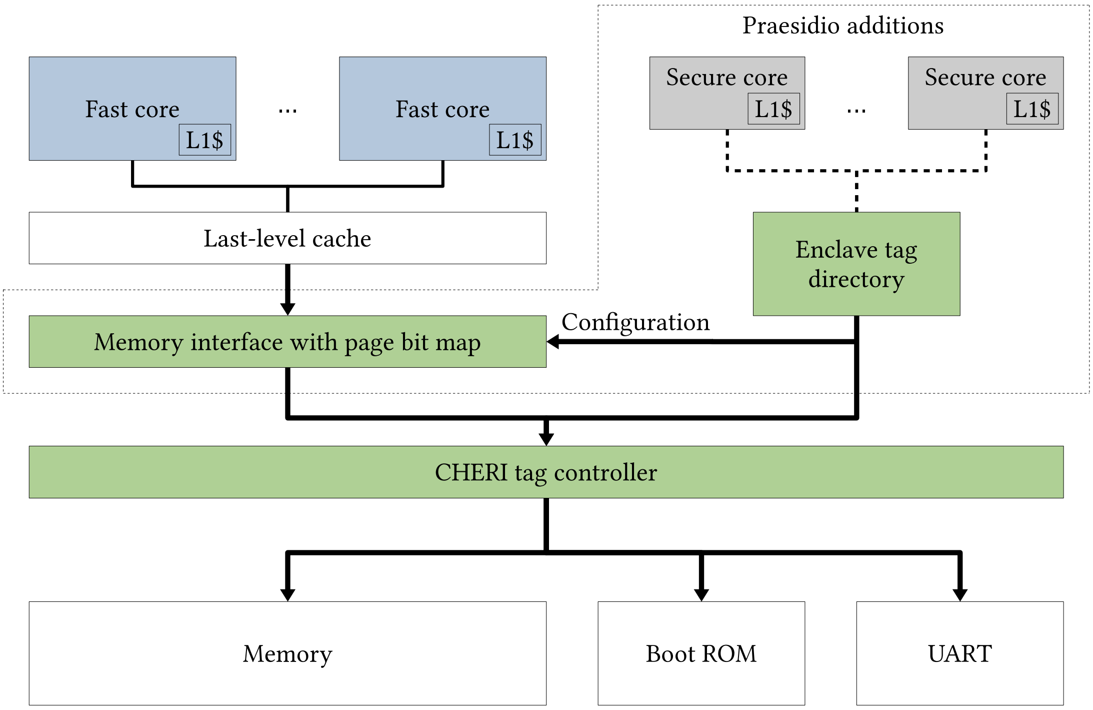

Figure 7.1: The new memory hierarchy on which the hardware evaluation is based, with the Praesidio additions contained within the dashed lines. The green boxes are the inherently trusted parts of the system to perform memory access control. The secure cores, shown as gray boxes, are only trusted when they are running the management shim. Finally, the fast cores are completely untrusted and have a blue background. This figure shows how the LLC is now only usable by the fast cores so that inter-core side-channel attacks are no longer possible through the LLC. Additionally, this allows me to move the memory interface for the fast cores to after the LLC, and this should minimize the performance cost of the access control for the fast cores.

provide security for enclaves is possible without significant performance loss. This is essential to consider my approach as an alternative to adding enclave features to existing fast cores.

7.2 Hardware implementation overview

Figure 7.1 shows the high-level architecture of how to connect this memory interface in the new hardware system. It shows how the last-level cache (LLC) is now only used for fast cores and how the memory interface moves from between the first-level cache and the LLC to between the LLC and dynamic random-access memory (DRAM).

As a fast core, I use Toooba, which is a superscalar out-of-order RISC-V processor [116]. This open-source processor is based on MIT's RiscyOO processor [117]. The main addition I make to this processor is a memory interface, which enforces access control. This memory interface ensures that when the fast core donates memory to an enclave, that piece of memory becomes inaccessible to the fast core; accesses are blocked and return all bits set to 1 [13]. The memory interface only restores the fast core's access to that memory when the enclave has been deleted and its data has been cleared from memory.

One optimization that I explore in this chapter is to handle all the access control in the memory interface. This allows me to use only one pair of bits per page, rather than having to use larger multi-bit tags. One bit in the pair indicates whether the fast core owns that page, and the other bit indicates whether it has just read access to that page. Assuming that the DRAM is 1 GiB in size and pages are 4 KiB, this means that the size of this bit mask is 1,024 × 1,024 × 2/(4 × 8) B = 64 KiB. This is a reasonable size compared to an example size of 512 KiB for the LLC, which is the same as in BOOM [2]. This bit mask is the same for all fast cores because fast cores all run using the default enclave identifier.

To evaluate the impact that this system has on the performance of the fast cores, I initially only need to implement the memory interface for the fast cores. I call this interface the fast memory interface (FMI).

7.3 Fast memory interface

In this section I describe the implementation of the FMI. To evaluate this FMI, I use a prototype system with a single fast core and no secure cores. This prototype allows me to evaluate the performance cost of implementing this enclave system on the fast cores, independent of whether any enclaves are running.

A CHERI-enabled Toooba introduces a CHERI tag controller (CTC) between the LLC and the main advanced extensible interface (AXI) bus. The CTC pairs an out-of-band validity bit with each 128-bit piece of memory [52]. This validity bit indicates whether that memory contains a valid capability or not. My enclave access control works orthogonal to this protection methodology. I propose adding a memory interface in between the LLC and the CTC. It is important that the CTC comes after the FMI and also covers the secure cores, so that CHERI validity tags stay consistent between fast and secure cores. Overlooking this can lead to tags being visible to some cores but not to others, which can lead to forged capabilities and would break CHERI's security assumptions.

Originally the LLC's AXI manager[1] port is connected to the CTC's subordinate port. The idea is to add an FMI in between these two modules. The FMI also has an AXI manager and subordinate port. The FMI can be slotted in between the LLC and the CTC by connecting the LLC's manager port to the FMI's subordinate port and the FMI's manager port to the CTC's subordinate port.

Besides these two ports, the FMI also has a configuration port, which is used to set the bitmap controlling which pages are accessible by the fast cores. The bitmap contains two bits for every page. The first bit indicates whether the default enclave owns this page, and the second bit indicates whether the default enclave has read-write or read-only access. This configuration port is an AXI subordinate port and is connected to the output of the tag directory so that it can only receive requests originating from a secure core. The configuration of the FMI is memory mapped. Figure 7.2 shows a listing of the Bluespec definition of the FMI's interface.

The FMI should by default deny access to all pages until it is configured by a secure core. This allows the management shim to initialize itself and clear any sensitive memory before the fast cores can start to interact with memory.

The FMI adds a single cycle of latency to memory requests to DRAM, which on a regular central processing unit (CPU) takes at least 50 cycles (see Section 7.4.1 for how this is calculated). This cycle is necessary to consult the local block random-access memory (BRAM) in which the bitmap is stored. Responses from DRAM can be forwarded without delay.

To connect the FMI, the original core wrapper (`CoreW.bsv`) is wrapped in a larger wrapper (`Praesidio_CoreWW.bsv`) [119]. The Praesidio wrapper instantiates both the original core wrapper and the FMI. It merges the cached and uncached requests from the cores and funnels them through the FMI. Finally, I connect the FMI's configuration port, which in this prototype is connected to the main AXI bus since there are no secure cores in this system yet. I also specify which address range the configuration is mapped at.

7.3.1 Evaluation

This evaluation is done using a version of Toooba published by the Computer Architecture group at the University of Cambridge [120]. The framework used to build Toooba is developed by Galois for the system security integration through hardware and firmware program from DARPA. Adding the FMI leaves the critical path unaltered when synthesizing for the VCU118 FPGA [121]. This means that it can be incorporated in a system on chip (SoC) without decreasing the frequency of the processor. In terms of area, including the FMI insignificantly increases

[1] In January 2021, the AMBA AXI and ACE protocol specification "regularized terminology to use Manager to indicate the agent that initiates transactions and Subordinate to indicate the agent that receives and responds to requests" [118].

```
interface Praesidio_MemoryShim #(
    numeric type id_,
    numeric type cid_,
    numeric type addr_,
    numeric type data_,
    numeric type awuser_,
    numeric type wuser_,
    numeric type buser_,
    numeric type aruser_,
    numeric type ruser_);

  method Action clear;

  // Manager port to connect to main AXI4 bus towards
  //   DRAM etc.
  interface AXI4_Manager #(
    id_,  addr_, data_, awuser_, wuser_, buser_,
    aruser_, ruser_
  ) manager;

  // Subordinate port to receive memory requests from
  //   core
  interface AXI4_Subordinate #(
    id_,  addr_, data_, awuser_, wuser_, buser_,
    aruser_, ruser_
  ) subordinate;

  // Subordinate port to connect to main AXI4 bus for
  //  configuration
  interface AXI4_Subordinate#(
    cid_, addr_, data_, awuser_, wuser_, buser_,
    aruser_, ruser_
  ) configSubordinate;
endinterface
```

Figure 7.2: Listing that shows the code of the Bluespec interface for the FMI, which has two ports for connecting into the memory subsystem and one configuration port.

```
$ time sh -c `dd if=/dev/random of=/tmp/rand bs=8M count=1; \
              bzip2 /tmp/rand; bunzip2 /tmp/rand.bz2'
```

Figure 7.3: The bash commands run on FreeBSD as a micro-benchmark to test the memory access time with and without the FMI.

the number of look-up tables (LUTs) used. In fact in my synthesis it decreases it slightly by 0.2%, which means that the increase from adding the extra hardware was insufficient to offset the random variation that synthesis tools have when generating designs with different seeds. The main area cost is the memory used by the bitmap, which increases the BRAM usage on the FPGA by by about 3.8%.

To compare a single-core Toooba implementation with and without the FMI, I run a memory-intensive micro-benchmark on FreeBSD. Namely, I take some random data, compress that data and then decompress it with the commands shown in Figure 7.3. The average time over three runs without FMI is 495.7 seconds and with FMI this increases to 505.5 seconds. Comparing these times shows and increase of 1.98% ± 0.08 in terms of runtime.

There are numerous variables involved with running such a benchmark on FreeBSD. For example on some Linux distribution `/dev/random` might block depending on how much entropy is available on the system, which is not the case for FreeBSD. This is why I develop an alternative way to measure this overhead. The alternative measure is a bare-metal benchmark, written in RISC-V assembly, that loads from memory that is not yet in the cache and use that value from memory to determine the next load address. This way each load is dependent on the previous load. The assembly code loops over this process of loading from an uncached location and calculating a new uncached location based on the loaded value for 1,000 iterations. Running this benchmark on an unmodified CHERI Toooba core takes 58,606 cycles, while adding the FMI increases this to 59,580, which is an increase of 1.66%. Most applications running on fast cores will experience less slowdown because caches mask the DRAM access time in the common case.

7.3.2 Optimizations

Currently each page takes 2 bits of encoding to encode 3 possible configurations. A system can compress this by using more bits to encode the ternary permission values for multiple pages. Theoretically, if the number of encoded pages per entry approaches infinity, each ternary value can be encoded in $\log_2 3$ bits, so the optimal savings are $\frac{2-\log_2 3}{2} = \frac{\log 4/3}{\log 4} \approx 20.75\%$. Table 7.1 shows the savings made by increasing the number of pages per entry. Savings can go up to at least 0.4150 bits per page or 20.75% for 79,335 pages, but it would require a huge bus to transport the 125,743 bits necessary to encode these values. However, encoding 41 pages

Pages per Entry	Entry Size (bits)	Savings per Page (bits)	BRAM Size Reduction (%)
1	2	0.00000	0.000
3	5	0.33333	16.667
5	8	0.40000	20.000
17	27	0.41176	20.588
29	46	0.41379	20.690
41	65	0.41463	20.732
94	149	0.41489	20.745
147	233	0.41497	20.748
200	317	0.41500	20.750
253	401	0.41502	20.751
306	485	0.41503	20.752
971	1539	0.41504	20.752

Table 7.1: Compressed mappings from FMI bitmap entries to page permissions.

in 65 bits gets most of the savings, while keeping the entry size in the BRAM reasonable. Compared to the optimum, using a 65 bit entry size would only increase the BRAM size by 106 bits per gigabyte of DRAM. Saving approximately 20% on a memory that is a few kilobytes large is a small optimization when thinking about the amount of memory that is on a modern SoC, but since I am already using 64 bit entry sizes, it seems like low-hanging fruit to do so by increasing the entry size by 1 bit. Another option is using an entry size of 8, which is a power of two and still achieves a size reduction of 20%.

Besides the size optimization, a system can also reduce the average memory access latency by doing the bitmap lookup in parallel to LLC requests. As long as this parallel lookup does not affect the overall LLC performance, this should remove the overhead completely with the downside of increasing power consumption because each hit in the LLC now also requires a lookup.

7.4 Secure cores

To get a better understanding of how Praesidio affects the performance of a hardware implementation, I add a secure core to the SoC. These secure cores should be in-order cores without simultaneous multithreading to avoid intra-core side-channel attacks (see Section 5.7.3). In this prototype I use a secondary Toooba core because it was infeasible to connect an in-order core to the same memory hierarchy. This still allows me to measure performance of the memory hierarchy even if the CPU performance is slightly different from what a final system would have. More specifically I measure the communication latency between fast and secure cores as well as the area overhead of adding the secure cores and their caches.

To estimate the latency that a tag directory would impose on secure cores, I add a secondary FMI in front of the secure cores. This produces the same delay that a tag directory would have, since both have to perform a BRAM lookup and this is what causes the delay. Using a single FMI for the secure world maintains the security guarantees if page table management becomes part of the management shim's trusted computing base (TCB). In Chapter 4 I left page table management outside of the TCB because each enclave can then delegate this to either another enclave or to their own chosen enclave runtime. However, this does not have to be the case and this is a trade-off between complexity of the software TCB and the hardware TCB. For the rest of this evaluation I use an FMI for the secure world instead of a full-fledged tag directory.

7.4.1 Communication overhead

Communication between normal applications and enclaves incurs some latency due to the security measures taken in the memory hierarchy. Taking the example of an application running on a fast core that wants an enclave to sign the contents of a page. The application must set the page's reader tag to the appropriate identifier so that the correct enclave is able to read data from it. Once this is done the application must send the page address to the enclave. This operation incurs latency due to the system call required in the normal world to update the reader tag, as well as the round-trip latency between the application and the enclave. I measure the round-trip time between the fast cores and the secure cores by starting a timer in a fast core, writing a word from that fast core, waiting to see that value in a secure core, writing a different value to that word from that secure core, waiting to see that value from the fast core and stopping the timer. The latency is primarily caused by uncached accesses to DRAM, since Praesidio prohibits the sharing of caches between fast cores and secure cores to avoid cache-based side channels.

Modern FPGAs include dedicated DRAM chips, which make them accessible within a few cycles of a processor running on the FPGA's LUTs. To better model the DRAM latency a processor would experience on a normal SoC, I take the column address strobe (CAS) latency of a Samsung DDR4 module, which is 11 cycles on a 0.8 GHz clock [31]. The CAS latency is the time it takes to access an open row in DRAM, which is the minimum latency for accessing any memory in DRAM. To translate this into processor cycles, I take the frequency of a desktop Intel core i5-7500 CPU from the same era, which is 3.4 GHz [122]. To calculate the CAS latency in CPU cycles, I use the following equation:

$$C_{\text{cpu}} = \frac{C_{\text{dram}}}{F_{\text{dram}}} \times F_{\text{cpu}} = \frac{11}{0.8} \times 3.4 = 47 \text{ cycles, where } C \text{ is cycles and } F \text{ is frequency.}$$

To measure the communication latency overhead, I introduce an extra latency of 47 cycles between the AXI bus and DRAM. I then run an experiment on a baseline system and a system using Praesidio. The baseline system measures communication latency between two Toooba cores connected to the same cache hierarchy. In this scenario an average round-trip latency of 186 cycles is observed. The system using Praesidio attempts to perform the same communication between cores, but without having a cache hierarchy to talk through. In the scenario with Praesidio, an average latency of 264 cycles is observed.

I calculate that the round-trip time is increased by 42%. This is consistent with my expectations, since communication now has to go all the way out to DRAM and back rather than stopping at the LLC. It is also important to realize that this round-trip time does not affect the throughput of an enclave — it becomes relevant only when communicating between the fast core and enclaves, or between separate enclaves. Since secure core throughput is unaffected by this, Praesidio can minimize performance impact by pipelining requests; when a fast core wants to make back-to-back signing requests, it can send the next request before it has received the previous response. This way there is still processing occurring while the message makes its way through the memory hierarchy, which helps to mask the latency. If the round-trip communication latency is still a significant bottleneck, then it might be desirable to introduce more intrusive architectural changes. One possibility would be a shared LLC as discussed in Chapter 4 and depicted in Figure 4.1. This would bypass the fixed CAS latency that is required for an access to DRAM.

7.4.2 Area overhead

Another important metric for the evaluation of any hardware system is the area overhead. The area overhead for Praesidio originates in the modifications to the memory hierarchy and the addition of secure cores. To measure this overhead I use two CHERI-enabled processors developed at the architecture group of the Computer Laboratory at the University of Cambridge. The processor used as the application processor in this evaluation is the Toooba core [120]. Toooba is a superscalar and out-of-order core designed to run performance-intensive applications. Although in the earlier experimental setup I did not use an in-order core, for the area overhead I can use a separate synthesized version of Flute to extract the area of just the core logic as an estimate for the secure cores [123]. Flute is a 5-stage in-order core meant more for embedded-style systems, but which can be parameterized to support more privilege modes, 64 bit computing and several RISC-V extensions. In this evaluation I consider a parameterization of Flute that features the same RISC-V extensions as Toooba; hence the same computation can be performed on Flute as on Toooba. This means that the values for overheads observed here should be seen as an upper limit, and can be reduced by using a different parameterization of Flute, perhaps without a dedicated floating point pipeline or fewer privilege levels. When built for the VCU118 FPGA, Flute takes about 76,900 LUTs, while Toooba takes 391,600 LUTs. These area numbers are for just the core and its caches so excluding the rest of the SoC. Flute is thus a factor of 5 smaller than Toooba and so adding 5 secure cores costs the same area as adding 1 fast core.

In addition to logic blocks, Toooba also takes up more memory blocks because it has larger cache sizes. Flute has an 8 KiB first-level cache with no other caching. For Toooba each core has a 32 KiB first-level cache, and there is also a shared 1 MiB LLC. This size difference is reflected in the BRAM usage — Flute uses 38 BRAM blocks and Toooba uses 294. I expect secure cores to be less geared towards performance, and thus smaller caches are not a problem. Adding a secure Flute core to a system containing a Toooba core increases the LUTs used by 19.6% and the number of BRAM blocks used by 12.9%.

Taken in isolation, this upper limit to the increase in hardware seems to be worthwhile to empower application developers with higher security execution environments. This becomes more appealing considering that the dominant factor in silicon performance is power rather than silicon area in the age of dark silicon [93]. For this reason accelerators are becoming more prevalent in SoCs, so it is reasonable to expect that manufacturers would be willing to use some extra silicon area to provide better security.

7.5 Discussion

Although Section 2.6 provides compelling use cases for secure cores, we still needed to show that Praesidio is a worthy alternative approach to securing fast cores. Securing fast cores (like Intel SGX [13] and Arm realm management extension [15]) will inevitably lead to challenging security versus performance trade-offs and cause extra complications in the design process. Praesidio moves this complexity to designing dedicated secure cores and designing a secure memory hierarchy. However, this argument only holds up if Praesidio itself does not slow down the fast cores, and that is exactly what this chapter shows. This does *not* mean that Praesidio is better than securing fast cores, but it does mean that physical isolated enclaves are a worthy point on the design spectrum and a realistic alternative to current enclave system designs.

7.6 Summary

In this chapter I present a hardware implementation of the Praesidio system to better understand the performance impact it has in a real world system. Adding the FMI to the memory hierarchy of the fast cores slows down these application cores by around 2% for memory intensive workloads. This is likely to be an overestimate due to DRAM being relatively faster on FPGA than in real-world systems. Communication latency between secure cores and fast cores is acceptable with the round-trip overhead increasing by 42% and the area overhead of adding secure cores comes at about a fifth of the logic of the fast cores. All in all this hardware-based evaluation backs up the evaluation made in Chapter 6 to show that Praesidio is a viable security system.

Chapter 8

Conclusion

This thesis shows how to improve the security of applications running on modern central processing units (CPUs). Particularly some applications need protection from privileged code running on the same system. Previous work has generally taken two approaches: implementing isolation on a modern processor or introducing a completely separate secure system on chip (SoC). The benefit of the first approach is that secure applications can still use the high performance core, whereas the separate secure SoC has better security isolation properties.

I define an enclave threat model that includes side-channel attacks which can be performed by privileged software (Section 2.8). Adding such enclave functionality to a fast application core is unlikely to be successful: it will either prohibitively slow down applications or provide insufficient protection to enclaves. Fast application cores contain a huge amount of predictive microarchitectural state: caches, branch targets, value prediction, address prediction, store buffers, pre-fetchers, etc. This predictive state is good for performance, but it also makes it harder to protect assets from leaking through side channels. I hypothesize that protecting enclaves from privileged code, while running both of these on a main application core, is infeasible without employing expensive fence operations. It is undesirable to use these fences on fast cores frequently without degrading the runtime performance of normal applications. Praesidio avoids this by only needing to perform secure context switches and fences on separate secure cores. Physically isolating enclaves is similar to introducing a dedicated separate secure SoC (like a trusted platform module), but still allows the rich operating system (OS) on the fast application core to perform resource management. Additionally, physically isolating enclaves is possible while keeping efficient and secure access to dynamic random-access memory (DRAM).

On the way to exploring side-channel attacks on enclaves, I define a methodology for converting side channels into direct channels that quantifies an upper bound on the possible information leakage through such channels. If the direct channel is closed, then the side channel is closed as well (Chapter 3). To adhere to the new enclave threat model, I propose a system design with secure cores for enclaves to protect them from intra-core side-channel attacks without stringent security requirements on the design of fast application cores (Chapter 4). I provide a security analysis to demonstrate that physically isolating enclaves and carefully designing a memory hierarchy can protect against all the threats in my threat model (Chapter 5). For evaluation I implement a Linux driver, a management shim, an instruction set simulator and a hardware implementation of a physically isolated system (Chapters 6 and 7). The evaluation shows that these protection mechanisms create:

- a similar number of cache accesses to communicating over Unix pipes,
- an increase of less than 2% in runtime for memory intensive workloads on the fast cores,
- a 42% increase in communication latency between application and enclave compared to two applications running on separate fast cores,
- a negligible increase in DRAM size,
- a modest increase of 2% to 12% in last-level cache (LLC) size,
- a hardware area increase of between 2% and 14% based on an existing big-little architecture
- and an increase of 20% in look-up tables (LUTs) and 13% in block random-access memory (BRAM) for a field-programmable gate array (FPGA) implementation.

All in all, these numbers mean that the performance impact on the fast cores is insignificant. The communication latency between fast and secure cores is significant, but this overhead does not affect the performance of the rest of the system like speculative barriers do. Additionally, this latency is amortized by use cases that send larger data to enclaves like for signing messages.

8.1 Future work

This thesis also uncovers future avenues of work, such as designing a dedicated secure core as an enclave processor. This core needs to be carefully designed to avoid side-channel attacks and can be aided by using the direct channel methodology from Chapter 3. The secure cores should ideally be small so that their isolation properties can be formally verified. They should also only have predictive state that is verifiably compartmentalized between enclaves.

As an extension of this, heterogeneous secure cores that allow for multidimensional performance and security trade-offs are a promising area to explore. For example a throughput core that has high side-channel protection, medium multi-thread performance and *low single-thread performance;* or a simple in-order core that has high side-channel protection, *low multi-thread performance* and medium single-thread performance. Additionally, there is a fast application core that has *low side-channel protection,* high multi-thread performance and high single-thread performance. There are three dimensions here: side-channel protection, multi-thread performance and single-thread performance. System designers can conceivably create a vast array of cores that lie anywhere on these axes. There are also many more dimensions that can be added for interesting future work, like memory throughput, splitting side-channel protection into multiple dimensions, etc.

Having more efficient solutions to closing the timing channels in the memory hierarchy would also be beneficial. Closing the DRAM timing channel without using a constant access time in the DRAM controller would improve performance. Characterizing all the different timing channels through the LLC and the memory bus is a significant project, as well as proving that the variation in access time does not leak any data about memory accesses to outside the enclave.

Verifying the trusted kernel that I call the management shim is another piece of crucial future work before this technology can be deployed in the field. In this thesis I show that it is possible to create a complete enclave system that physically isolates enclaves on secure cores while keeping the trusted computing base (TCB) to a minimum. Namely, the management shim is simpler and an order of magnitude smaller than seL4. It should be possible to prove the security properties of the management shim using the same formal method techniques.

Finally, fine-tuning the optimal properties of a final system, such as how many secure cores are needed and whether they should focus more on multi-thread or single-thread performance, is still an open problem. The main challenge with finalizing such properties is that there is a chicken and egg problem; we need enclave use cases to finalize the properties, but these use cases will never be developed until a compelling system exists. The platform that I present in this thesis is hopefully the first step in solving this chicken and egg problem. Developers can prototype enclaves on this system, which can then be used to define the properties of future enclave systems.

8.2 Open problems

As discussed in Section 2.10 this thesis tackles a subset of the research challenges in this area. This section discusses some of the open problems that still need to be tackled.

Cache compartmentalization is non-trivial. I show that existing compartmentalization schemes work with Praesidio, but it is unclear which one is best. There is a trade-off between security and flexibility. For now the conservative approach of static partitioning seems to be the consensus within research, but this is not used in industrial systems. The open problem is deciding whether a flexible partitioning scheme is realistic from a security standpoint. Additionally, it is possible to have a cache be partially statically and partially flexibly partitioned. All of these options need to be evaluated and studied.

One way in which adding enclave functionality to fast cores can win out over physically isolating enclaves is if a fast core can be proven secure in terms of isolation from direct attacks, intra-core side-channels attacks and transient execution attacks. If a fast core with enclave functionality can be proven correct while preserving acceptable performance levels, this may make it unnecessary to have a heterogeneous system based on core security. If this open problem is solved, it can conclude the debate about physically isolated enclaves.

Another approach that can change the debate on physically isolated enclaves is whether it is possible to run the lightweight compartment use cases on existing physically isolated systems. Currently most physically isolated systems have the restriction of being a walled garden, on which only vendor trusted software is allowed to run. If we can run mutually distrusting software on these systems, this might be exactly the type of physically isolated enclave system we need to fulfill the security requirements in already deployed systems.

The area of physically isolating lightweight compartments is rich and there are multiple open problems that still need to be covered. This thesis shows one way of tackling physically isolating enclaves and shows that it is a realistic alternative to current enclave systems.

8.3 Final words

I came into this PhD with an interest in privacy and how applications can keep data private from the OS. I soon realized that side-channel attacks were an overlooked area, and shortly after Spectre and Meltdown shook the computer architecture world. In the end my research explores aspects of protecting enclaves from privileged code, even in the face of side-channel attacks. I hope this work moves this area forward in both research and industry to empower users and developers with better security and privacy in their computing systems.

Bibliography

[1] M. van der Maas and S. W. Moore, "Protecting enclaves from intra-core side-channel attacks through physical isolation," in *Proceedings of the 2nd Workshop on Cyber-Security Arms Race*, CYSARM'20, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2020.

[2] U. of California, "The Berkeley out-of-order machine (BOOM)." https://docs.boom-core.org/en/stable/, 2019.

[3] "Intel 64 and IA-32 architectures optimization reference manual." https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2016.

[4] A. Frumusanu, "The iPhone XS & XS Max review: Unveiling the silicon secrets." https://www.anandtech.com/show/13392/the-iphone-xs-xs-max-review-unveiling-the-silicon-secrets/, 2018.

[5] "Introduction to CHERI," Tech. Rep. UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, September 2019.

[6] A. Waterman and K. Asanović, "The RISC-V instruction set manual, volume I: Unprivileged ISA," December 2019. Version 20191213.

[7] S. Biggs, D. Lee, and G. Heiser, "The jury is in: Monolithic OS design is flawed: Microkernel-based designs improve security," in *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys)*, pp. 16:1–16:7, ACM, 2018.

[8] P. G. Neumann, "Principled assuredly trustworthy composable architectures," tech. rep., Computer Science Laboratory, SRI International, Menlo Park, California, December 2004. http://www.csl.sri.com/neumann/chats4.pdf.

[9] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone explained: Architectural features and use cases," in *2nd IEEE International Conference on Collaboration and Internet Computing (CIC)*, pp. 445–451, IEEE Computer Society, 2016.

[10] "Apple T2 security chip security overview." https://www.apple.com/jp/mac/docs/Apple_T2_Security_Chip_Overview.pdf, October 2018.

[11] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (J. N. Matthews and T. E. Anderson, eds.), pp. 207–220, ACM, 2009.

[12] A. D. Zonenberg and B. Yener, "Antikernel: A decentralized secure hardware-software operating system architecture," *IACR Cryptology ePrint Archive*, p. 550, 2016.

[13] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, p. 86, 2016.

[14] D. Kaplan, J. Powell, and T. Woller, "AMD: Memory encryption." https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, April 2016.

[15] "Arm realm management extension (RME) system architecture," Tech. Rep. DEN 0129 A.a, Arm Limited, June 2021.

[16] C. Canella, J. van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," *CoRR*, vol. abs/1811.05441, 2018.

[17] T. Bourgeat, I. A. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 42–56, ACM, 2019.

[18] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. Lai, "SgxPectre attacks: Leaking enclave secrets via speculative execution," *CoRR*, vol. abs/1802.09085, 2018.

[19] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *Proceedings of the 22nd USENIX Security Symposium* (S. T. King, ed.), pp. 479–494, USENIX Association, 2013.

[20] "Intel trust domain extensions," Tech. Rep. 343961-001US, Intel Corporation, August 2020.

[21] B. Schneier and A. Shostack, "Breaking up is hard to do: Modeling security threats for smart cards," in *Proceedings of the 1st Workshop on Smartcard Technology* (S. B. Guthery and P. Honeyman, eds.), USENIX Association, 1999.

[22] W. Arthur and D. Challener, *A Practical Guide to TPM 2.0.* London: Apress, 2015. Using the Trusted Platform Module in the New Age of Security.

[23] D. E. Bell and L. LaPadula, "Secure computer system: Unified exposition and Multics interpretation," Tech. Rep. ESD-TR-75-306, The MITRE Corporation, March 1976.

[24] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Asia Pacific Workshop on Systems (APSys)* (H. Chen, Z. Zhang, S. Moon, and Y. Zhou, eds.), p. 5, ACM, 2011.

[25] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," *IACR Cryptology ePrint Archive*, p. 271, 2005.

[26] J. D. Golic, "Multiplicative masking and power analysis of AES," *IACR Cryptology ePrint Archive*, p. 91, 2002.

[27] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium* (T. Holz and S. Savage, eds.), pp. 857–874, USENIX Association, 2016.

[28] D. Kaplan, J. Powell, and T. Woller, "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, January 2020.

[29] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A. Sadeghi, "TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V," in *26th Annual Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2019.

[30] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *25th USENIX Security Symposium* (T. Holz and S. Savage, eds.), pp. 565–581, USENIX Association, 2016.

[31] "Samsung 4Gb E-die DDR4 SDRAM," Tech. Rep. K4A4G045WE, January 2017. Rev. 1.6.

[32] Y. Kim, R. Daly, J. Kim, C. Fallin, J. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 361–372, IEEE Computer Society, 2014.

[33] M. Kim, J. Choi, H. Kim, and H. Lee, "An effective DRAM address remapping for mitigating rowhammer errors," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1428–1441, 2019.

[34] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 570–580, 2007.

[35] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *36th IEEE Symposium on Security and Privacy (S&P)*, pp. 640–656, IEEE Computer Society, 2015.

[36] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," *IACR Cryptology ePrint Archive*, p. 1060, 2018.

[37] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," *CoRR*, vol. abs/1903.01843, 2019.

[38] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via Intel/AMD micro-op caches," in *48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pp. 361–374, IEEE, 2021.

[39] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," *CoRR*, vol. abs/1611.06952, 2016.

[40] D. Evtyushkin, R. Riley, N. B. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A new side-channel attack on directional branch predictor," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, eds.), pp. 693–707, ACM, 2018.

[41] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level directional predictor based side-channel attack against SGX," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, no. 1, pp. 321–347, 2020.

[42] J. van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking transient execution through microarchitectural load value injection," in *41st IEEE Symposium on Security and Privacy (S&P)*, pp. 54–72, IEEE, 2020.

[43] D. Evtyushkin, J. Elwell, M. Ozsoy, D. V. Ponomarev, N. B. Abu-Ghazaleh, and R. Riley, "Iso-X: A flexible architecture for hardware-managed isolated execution," in *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 190–202, IEEE Computer Society, 2014.

[44] A. Chaudhry, J. Crowcroft, H. Howard, A. Madhavapeddy, R. Mortier, H. Haddadi, and D. McAuley, "Personal data: Thinking inside the box," in *Proceedings of The Fifth Decennial Aarhus Conference on Critical Alternatives* (O. W. Bertelsen, K. Halskov, S. Bardzell, and O. Iversen, eds.), pp. 29–32, Aarhus University Press / ACM, August 2015.

[45] S. Zeitouni, J. Vliegen, T. Frassetto, D. Koch, A. Sadeghi, and N. Mentens, "Trusted configuration in cloud FPGAs," in *29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 233–241, IEEE, May 2021.

[46] "Microsoft PlayReady content protection technology." https://www.microsoft.com/playready/documents/, 2015.

[47] "Moving to HTML5 video." https://developers.google.com/media/pdf/2016-HTML5-Video-Whitepaper.pdf, 2016.

[48] F. Alliance, "Client to authenticator protocol (CTAP) – proposed standard." https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html, January 2019.

[49] S. Tamrakar, J. Liu, A. Paverd, J. Ekberg, B. Pinkas, and N. Asokan, "The circle game: Scalable private membership test using trusted hardware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (AsiaCCS)* (R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, eds.), pp. 31–44, ACM, April 2017.

[50] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[51] Hyperledger, "How Walmart brought unprecedented transparency to the food supply chain with Hyperledger fabric," 2019.

[52] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann,

A. Mazzinghi, A. Richardson, S. D. Son, and A. T. Markettos, “Efficient tagged memory,” in *35th IEEE International Conference on Computer Design (ICCD)*, pp. 641–648, IEEE Computer Society, 2017.

[53] D. Lie, C. A. Thekkath, and M. Horowitz, “Implementing an untrusted operating system on trusted hardware,” *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 178–192, 2003.

[54] G. E. Suh, C. W. O’Donnell, and S. Devadas, “Aegis: A single-chip secure processor,” *Information Security Technical Report*, vol. 10, no. 2, pp. 63–73, 2005.

[55] E. Owusu, J. Guajardo, J. M. McCune, J. Newsome, A. Perrig, and A. Vasudevan, “OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms,” in *2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (A. Sadeghi, V. D. Gligor, and M. Yung, eds.), pp. 13–24, ACM, 2013.

[56] D. Champagne and R. B. Lee, “Scalable architectural support for trusted software,” in *16th International Conference on High-Performance Computer Architecture (HPCA)* (M. T. Jacob, C. R. Das, and P. Bose, eds.), pp. 1–12, IEEE Computer Society, 2010.

[57] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, “Innovative technology for CPU based attestation and sealing,” in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, p. 7, Citeseer, 2013.

[58] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, “Keystone: An open framework for architecting trusted execution environments,” in *15th EuroSys Conference* (A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. Seltzer, eds.), pp. 38:1–38:16, ACM, 2020.

[59] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, “A formal foundation for secure remote execution of enclaves,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’17, (New York, NY, USA), p. 2435–2450, Association for Computing Machinery, 2017.

[60] “Cache speculation side-channels.” https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/, June 2020.

[61] S. W. Moore, R. Mullins, P. Cunningham, R. Anderson, and G. Taylor, “Improving smart card security using self-timed circuits,” in *8th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pp. 211–218, IEEE Computer Society, 2002.

[62] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *30th IEEE Symposium on Security and Privacy (S&P)*, pp. 45–60, IEEE Computer Society, May 2009.

[63] A. Nilsson, P. N. Bideh, and J. Brorsson, "A survey of published attacks on Intel SGX," *CoRR*, vol. abs/2006.13598, 2020.

[64] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[65] D. Evtyushkin, *Secure Program Execution Through Hardware-Supported Isolation.* PhD thesis, Graduate School of State University of New York at Binghamton, 2017.

[66] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, p. 91–98, May 2009.

[67] S. Ors, F. Gurkaynak, E. Oswald, and B. Preneel, "Power-analysis attack on an ASIC AES implementation," in *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, vol. 2, pp. 546–552 Vol.2, 2004.

[68] M. T. Rahman, Q. Shi, S. Tajik, H. Shen, D. L. Woodard, M. Tehranipoor, and N. Asadizanjani, "Physical inspection & attacks: New frontier in hardware security," in *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, pp. 93–102, 2018.

[69] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," *CoRR*, vol. abs/1812.08639, 2018.

[70] "Apple platform security." https://www.apple.com/ca/business/mac/pdf/aaw-platform-security.pdf, May 2021.

[71] D. Weston, "Meet the Microsoft Pluton processor – The security chip designed for the future of Windows PCs," November 2020.

[72] "Arm Corstone SSE-700 Secure Enclave," Tech. Rep. 101870_0000_03_en, July 2020. Rev. r0p0 EAC.

[73] F. Erata, S. Deng, F. Zaghloul, W. Xiong, O. Demir, and J. Szefer, "Survey of approaches and techniques for security verification of computer systems." IACR Cryptology ePrint Archive, 2016. https://ia.cr/2016/846.

[74] "Federal information processing standards publications (FIPS PUBS)." https://www.nist.gov/itl/publications-0/federal-information-processing-standards-fips, March 2022.

[75] "Common Criteria." https://www.commoncriteriaportal.org/, March 2022.

[76] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "IntroSpectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pp. 874–887, IEEE, June 2021.

[77] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Fuzzing for leaks in blackbox CPUs," *CoRR*, vol. abs/2105.06872, 2021.

[78] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing spectre-type vulnerabilities to the surface," in *29th USENIX Security Symposium* (S. Capkun and F. Roesner, eds.), pp. 1481–1498, USENIX Association, August 2020.

[79] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "DifFuzz: Differential fuzzing for side-channel analysis," in *Software Engineering 2020, Fachtagung des GI-Fachbereichs Softwaretechnik* (M. Felderer, W. Hasselbring, R. Rabiser, and R. Jung, eds.), vol. P-300 of *LNI*, pp. 125–126, Gesellschaft für Informatik e.V., February 2020.

[80] Y. Xiao, Y. Zhang, and R. Teodorescu, "SpeechMiner: A framework for investigating and measuring speculative execution vulnerabilities," *CoRR*, vol. abs/1912.00329, 2019.

[81] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *27th Annual Network and Distributed System Security Symposium, (NDSS)*, The Internet Society, February 2020.

[82] S. Jiang, Y. Ou, P. Pan, K. Cheng, Y. Zhang, and C. Batten, "PyH2: Using PyMTL3 to create productive and open-source hardware testing methodologies," *IEEE Design and Test*, vol. 38, no. 2, pp. 53–61, 2021.

[83] L. Domnitser, N. B. Abu-Ghazaleh, and D. Ponomarev, "A predictive model for cache-based side channels in multicore and multithreaded microprocessors," in *5th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, (MMM-ACNS)* (I. V. Kotenko and V. A. Skormin, eds.), vol. 6258 of *Lecture Notes in Computer Science*, pp. 70–85, Springer, September 2010.

[84] T. Zhang and R. B. Lee, "New models of cache architectures characterizing information leakage from cache side channels," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)* (C. N. P. Jr., A. Hahn, K. R. B. Butler, and M. Sherr, eds.), pp. 96–105, ACM, December 2014.

[85] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, “Microwalk: A framework for finding side channels in binaries,” *CoRR*, vol. abs/1808.05575, 2018.

[86] E. Prouff and M. Rivain, “Masking against side-channel attacks: A formal security proof,” in *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings* (T. Johansson and P. Q. Nguyen, eds.), vol. 7881 of *Lecture Notes in Computer Science*, pp. 142–159, Springer, 2013.

[87] H. Eldib, C. Wang, and P. Schaumont, “Formal verification of software countermeasures against side-channel attacks,” *ACM Transactions on Software Engineering Methodology*, vol. 24, no. 2, pp. 11:1–11:24, 2014.

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

[89] Z. Wang and R. B. Lee, “A novel cache architecture with enhanced performance and security,” in *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 83–93, IEEE Computer Society, 2008.

[90] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[91] N. Liu, W. Zang, S. Chen, M. Yu, and R. Sandhu, “Adaptive noise injection against side-channel attacks on ARM platform,” *EAI Endorsed Transactions on Security and Safety*, vol. 6, no. 19, p. e1, 2019.

[92] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, “Side-channel vulnerability factor: A metric for measuring information leakage,” in *39th International Symposium on Computer Architecture (ISCA)*, pp. 106–117, IEEE Computer Society, June 2012.

[93] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *38th International Symposium on Computer Architecture (ISCA)* (R. Iyer, Q. Yang, and A. González, eds.), pp. 365–376, ACM, 2011.

[94] W. Song, A. Bradbury, and R. Mullins, “Towards general purpose tagged memory,” *Proceedings of the RISC-V Workshop*, 2015.

[95] G. Platform, "TEE client API specification version 1.0." https://globalplatform.org/specs-library/tee-client-api-specification/, January 2017.

[96] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[97] M. Doblas, I.-V. Kostalabros, M. Moretó, and C. Hernández, "Enabling hardware randomization across the cache hierarchy in Linux-class processors," *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2020.

[98] M. M. Real, *Spatial Isolation against Logical Cache-based Side-Channel Attacks in Many-Core Architectures. (Isolation physique contre les attaques logiques par canaux cachés basées sur le cache dans des architectures many-core).* PhD thesis, University of Southern Brittany, Morbihan, France, 2017.

[99] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini, and G. Heiser, "Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core," *arXiv preprint arXiv:2005.02193*, 2020.

[100] E. Brickell and J. Li, "Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities," *IACR Cryptology ePrint Archive*, p. 194, 2007.

[101] E. Brickell and J. Li, "Enhanced privacy ID from bilinear pairing," *IACR Cryptology ePrint Archive*, p. 95, 2009.

[102] L. Xun and Z. Lili, "Intel SGX for Linux." https://github.com/intel/linux-sgx, 2022.

[103] C. F. Kerry, A. Secretary, and C. R. Director, "FIPS PUB 186-4 federal information processing standards publication digital signature standard (DSS)," 2013.

[104] Z. Sun, B. Feng, L. Lu, and S. Jha, "OEI: Operation execution integrity for embedded devices," *CoRR*, vol. abs/1802.03462, 2018.

[105] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)* (C. R. Ramakrishnan and J. Rehof, eds.), vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

[106] M. van der Maas, "Praesidio software." https://github.com/marnovandermaas/praesidio-software, 2020.

[107] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. M. Norton, and M. Roe, “The CHERI capability model: Revisiting RISC in an age of risk,” in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 457–468, IEEE Computer Society, 2014.

[108] N. Joly, S. ElSherei, and S. Amar, “Security analysis of CHERI ISA.” https://raw.githubusercontent.com/microsoft/MSRC-Security-Research/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf, 2020.

[109] L. G. Esswood, “CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor,” Tech. Rep. UCAM-CL-TR-961, University of Cambridge, Computer Laboratory, Sept. 2021.

[110] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, “TRRespass: Exploiting the many sides of target row refresh,” *CoRR*, vol. abs/2004.01807, 2020.

[111] A. Purnal and I. Verbauwhede, “Advanced profiling for probabilistic prime+probe attacks and covert channels in ScatterCache,” *CoRR*, vol. abs/1908.03383, 2019.

[112] M. van der Maas, “Praesidio SDK.” https://github.com/marnovandermaas/praesidio-sdk, 2020.

[113] S. Bush, “ARM’s Cortex-M0 processor – how it works.” https://www.electronicsweekly.com/news/products/micros/arms-cortex-m0-processor-how-it-works-2009-03/, 2009.

[114] M. Naylor, “POETS Twine.” https://github.com/POETSII/twine, 2018.

[115] A. Waterman and K. Asanović, “The RISC-V instruction set manual, volume II: Privileged architecture,” 2019. Version 1.11.

[116] R. S. Nikhil, J. Woodruff, and J. Clarke, “RISC-V core; superscalar, out-of-order, multicore capable; based on RISCY-OOO from MIT.” https://github.com/bluespec/Toooba, 2021.

[117] S. Zhang, A. Wright, T. Bourgeat, and Arvind, “Composable building blocks to open up processor design,” in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 68–81, IEEE Computer Society, 2018.

[118] “AMBA AXI and ACE protocol specification,” Tech. Rep. ARM IHI 0022H.c (ID012621), Arm Limited, January 2021.

[119] M. van der Maas, “Praesidio hardware modules.” https://github.com/marnovandermaas/praesidio-hardware, 2021.

[120] J. Woodruff, P. Rugg, J. Clarke, R. S. Nikhil, F. Fuchs, and M. van der Maas, “CHERI-enabled out-of-order RISC-V core.” https://github.com/CTSRD-CHERI/Toooba, 2021.

[121] Xilinx, *VCU118 Evaluation Board User Guide*, October 2018. https://www.xilinx.com/support/documentation/boards_and_kits/vcu118/ug1224-vcu118-eval-bd.pdf.

[122] “Intel® Core™ i5-7500 processor.” https://ark.intel.com/content/www/us/en/ark/products/97123/intel-core-i5-7500-processor-6m-cache-up-to-3-80-ghz.html, 2017.

[123] P. Rugg, R. S. Nikhil, J. Clarke, N. Sharma, and J. Woodruff, “CHERI RISC-V CPU, simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance.” https://github.com/CTSRD-CHERI/Flute, 2021.