Secure Processors Part II: Intel SGX Security Analysis and MIT Sanctum Architecture

Secure Processors Part II: Intel SGX Security Analysis and MIT Sanctum Architecture

Victor Costan, Ilia Lebedev and Srinivas Devadas
victor@costan.us, ilebedev@mit.edu and devadas@mit.edu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

now
the essence of knowledge
Boston — Delft

Foundations and Trends® in Electronic Design Automation

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

V. Costan, I. Lebedev and S. Devadas. *Secure Processors Part II: Intel SGX Security Analysis and MIT Sanctum Architecture*. Foundations and Trends® in Electronic Design Automation, vol. 11, no. 3, pp. 249–361, 2017.

This Foundations and Trends® issue was typeset in LaTeX using a class file designed by Neal Parikh. Printed on acid-free paper.

ISBN: 978-1-68083-302-7

Foundations and Trends® in Electronic Design Automation
Volume 11, Issue 3, 2017

Editorial Board

Editorial Scope

Topics

Foundations and Trends® in Electronic Design Automation publishes survey and tutorial articles in the following topics:

- System level design
- Behavioral synthesis
- Logic design
- Verification
- Test
- Physical design
- Circuit level design
- Reconfigurable systems
- Analog design
- Embedded software and parallel programming
- Multicore, GPU, FPGA, and heterogeneous systems
- Distributed, networked embedded systems
- Real-time and cyberphysical systems

Information for Librarians

Foundations and Trends® in Electronic Design Automation, 2017, Volume 11, 4 issues. ISSN paper version 1551-3939. ISSN online version 1551-3947. Also available as a combined paper and online subscription.

Foundations and Trends® in Electronic Design Automation
Vol. 11, No. 3 (2017) 249–361

DOI: 10.1561/1000000052

now
the essence of knowledge

Secure Processors Part II: Intel SGX Security Analysis and MIT Sanctum Architecture

Victor Costan, Ilia Lebedev and Srinivas Devadas
victor@costan.us, ilebedev@mit.edu and devadas@mit.edu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Contents

1 Introduction **3**

1.1 The Case for Hardware Isolation 4
1.2 Intel SGX is Not the Answer 5
1.3 MIT Sanctum Processor 6

2 An Analysis of Intel's Software Guard Extensions (SGX) **9**

2.1 SGX Implementation Overview 10
2.2 SGX Memory Access Protection 15
2.3 SGX Security Check Correctness 22
2.4 Tracking TLB Flushes 30
2.5 Enclave Signature Verification 34
2.6 Key Hierarchy and Derivation 39
2.7 SGX Security Properties 42

3 The MIT Sanctum Processor **61**

3.1 Threat Model 62
3.2 Programming Model Overview 64
3.3 Protection Boundaries 70
3.4 Security Primitives 70
3.5 Hardware Modifications 72
3.6 Software Design 79

3.7 Security Analysis of Sanctum 93
3.8 Work Related to Sanctum Mechanisms 103

4 Conclusion **105**

Acknowledgments **107**

References **109**

Abstract

This manuscript is the second in a two part survey and analysis of the state of the art in secure processor systems, with a specific focus on remote software attestation and software isolation. The first part established the taxonomy and prerequisite concepts relevant to an examination of the state of the art in trusted remote computation: attested software isolation containers (enclaves). This second part extends Part I's description of Intel's Software Guard Extensions (SGX), an available and documented enclave-capable system, with a rigorous security analysis of SGX as a system for trusted remote computation. This part documents the authors' concerns over the shortcomings of SGX as a secure system and introduces the MIT Sanctum processor developed by the authors: a system designed to offer stronger security guarantees, lend itself better to analysis and formal verification, and offer a more straightforward and complete threat model than the Intel system, all with an equivalent programming model.

This two part work advocates a principled, transparent, and well-scrutinized approach to system design, and argues that practical guarantees of privacy and integrity for remote computation are achievable at a reasonable design cost and performance overhead.

V. Costan, I. Lebedev and S. Devadas. *Secure Processors Part II: Intel SGX Security Analysis and MIT Sanctum Architecture*. Foundations and Trends® in Electronic Design Automation, vol. 11, no. 3, pp. 249–361, 2017.
DOI: 10.1561/1000000052.

1

Introduction

Between the Snowden revelations and the seemingly unending series of high-profile hacks of the past few years, the public's confidence in software systems has decreased considerably. At the same time, key initiatives such as cloud computing and the IoT (Internet of Things) are gaining popularity but require users to place much trust in the systems providing these services. We must therefore develop capabilities to build software systems with compelling security, and gain back our users' trust.

This manuscript is the second in a two part survey of the state of the art in secure processor systems, with a specific focus on remote software attestation and software isolation. Part I [Costan et al., 2017] established relevant background in computer system design (§ I.2) and security primitives (§ I.3), and surveyed relevant prior work (§ I.4). The same work discussed the attested software isolation container (enclave): a modern primitive for modular secure software and trusted remote computation, as exemplified by Intel's Software Guard Extensions (§ I.5).

This manuscript extends the discussion of enclaves and SGX by surveying the implementation and security properties of SGX (§ 2),

and documents the authors' concerns with its vulnerabilities to several classes of software attacks. Informed by the successes and shortcomings of SGX, this manuscript also discusses the MIT Sanctum processor (§ 3): a secure processor that offers an equivalent programming model with strong security guarantees against an insidious software threat model including cache timing and memory access pattern attacks. With this work, we hope to enable a shift in discourse in secure hardware architecture away from plugging specific security holes to a principled approach to eliminating attack surfaces.

1.1 The Case for Hardware Isolation

The best known practical method for securing a software system amounts to modularizing the system's code in a way that minimizes code in the modules responsible for the system's security. Formal verification techniques are then applied to these modules, which make up the system's trusted codebase (TCB). The method assumes that software modules are isolated, so the TCB must also include the mechanism providing the isolation guarantees.

Today's systems rely on an operating system kernel, or a hypervisor (such as Linux or Xen, respectively) for software isolation. However **each** of the last three years (2012-2014) witnessed over 100 new security vulnerabilities in Linux [cve, 2014a, Chen et al., 2011], and over 40 in Xen [cve, 2014b].

One may hope that formal verification methods can produce a secure kernel or hypervisor. Unfortunately, these codebases are far outside our verification capabilities: Linux and Xen have over *17 million* [Anthony, 2014] and 150,000 [xen, 2015] lines of code, respectively. In stark contrast, the seL4 formal verification effort [Klein et al., 2009] spent *20 man-years* to cover 9,000 lines of code.

Given Linux and Xen's history of vulnerabilities and uncertain prospects for formal verification, a prudent system designer cannot include either in a TCB (trusted computing base), and must look elsewhere for a software isolation mechanism.

Fortunately, Intel's Software Guard Extensions (SGX) [McKeen et al., 2013, Anati et al., 2013] has brought attention to the alternative of providing software isolation primitives in the CPU's hardware. This avenue is appealing because the CPU is an unavoidable TCB component, and processor manufacturers have strong economic incentives to build correct hardware.

1.2 Intel SGX is Not the Answer

Unfortunately, although the SGX design includes a vast array of defenses against a variety of software and physical attacks, it fails to offer meaningful software isolation guarantees. The SGX threat model protects against all direct attacks, but excludes "side-channel attacks", even if they can be performed via software alone.

Alarmingly, cache timing attacks require only unprivileged software running on the victim's host computer, and do not rely on any physical access to the machine. This is particularly concerning in a cloud computing scenario, where gaining software access to the victim's computer only requires a credit card [Ristenpart et al., 2009], whereas physical access is harder, requiring trespass, coercion, or social engineering on the cloud provider's employees.

Similarly, in many Internet of Things (IoT) scenarios, the processing units have some amount of physical security, but they run outdated software stacks that have known security vulnerabilities. For example, an attacker may exploit a vulnerability in an IoT lock's Bluetooth stack and obtain software execution privileges, then mount a cache timing attack on its access-granting process, and obtain the cryptographic key that opens the lock.

Furthermore, the analysis of SGX documentation as described in Part I of this work reveals that it is impossible for anyone but Intel to reason about SGX's security properties, because significant implementation details are not covered by the publicly available documentation. This is a concern, as the myriad of security vulnerabilities [Wojtczuk and Rutkowska, 2011, 2009b, Wojtczuk et al., 2009, Duflot et al., 2006, Rutkowska and Wojtczuk, 2008, Wojtczuk and Rutkowska,

2009a, Wecherowski, 2009, Embleton et al., 2010] in TXT [Grawrock, 2009], Intel's previous attempt at securing remote computation, show that securing the machinery underlying Intel's processors is incredibly challenging, even in the presence of strong economic incentives.

If a successor to SGX claimed to protect against cache timing attacks, substantiating such a claim would require an analysis of its hardware and microcode, and ensuring that no implementation detail is vulnerable to cache timing attacks. Barring a highly unlikely shift to open-source hardware from Intel, such analysis will never happen.

A concrete example: the SGX documentation [Int, 2013, 2014] does not state where SGX stores the EPCM (enclave page cache map). If the EPCM is stored in cacheable RAM, page translation verification is subject to cache timing attacks. Interestingly, this detail is unnecessary for analyzing the security of today's SGX implementation, as we know that SGX uses the operating system's page tables, and page translations are therefore vulnerable to cache timing attacks. The example does, however, demonstrate the fine nature of crucial details that are simply undocumented in today's hardware security implementations.

In summary, while the principles behind SGX have great potential, the SGX design does not offer meaningful isolation guarantees, and the SGX implementation is not open enough for independent researchers to be able to analyze its security properties.

1.3 MIT Sanctum Processor

The Sanctum processor's main contribution is a software isolation scheme that addresses the issues raised above: Sanctum's isolation provably defends against known software side-channel attacks, including cache timing attacks and passive address translation attacks. Sanctum is a co-design that combines *minimal* and *minimally invasive* hardware modifications with a trusted software security monitor that is amenable to rigorous analysis and does not perform cryptographic operations using keys.

Sanctum achieves minimality by reusing and lightly modifying existing, well-understood mechanisms. For example, Sanctum's per-

enclave page tables implementation uses the core's existing page walking circuit, and requires very little extra logic. Sanctum is minimally invasive because it does not require modifying any major CPU building block. It only adds hardware to the interfaces between blocks, and does not modify any block's input or output. The use of conventional building blocks limits the effort needed to validate a Sanctum implementation.

Sanctum demonstrates that memory access pattern attacks by malicious software can be foiled without incurring unreasonable overheads. Its hardware changes are small, small enough to present the added circuits, in their entirety, in Figures 3.9 and 3.10. Sanctum cores have the same clock speed as their insecure counterparts, as there are no modifications on the CPU core critical execution path. Using a straightforward page-coloring-based cache partitioning scheme with Sanctum adds a few percent of overhead in execution time, which is orders of magnitude lower than the overheads of the ORAM schemes [Goldreich, 1987, Stefanov et al., 2013] that are usually employed to conceal memory access patterns.

All layers of Sanctum's TCB are open-sourced [MIT, 2017], and unencumbered by patents, trade secrets, or other similar intellectual property concerns that would disincentivize security researchers from analyzing it. The Sanctum prototype targets the Rocket Chip [Lee et al., 2014], an open-sourced implementation of the RISC-V [Waterman et al., 2014, 2015] instruction set architecture, which is an open standard. Sanctum's software stack bears the MIT license.

To further encourage analysis, most of Sanctum's security monitor is written in portable C++ which, once rigorously analyzed, can be used across different CPU implementations. Furthermore, even the non-portable assembly code can be reused across different implementations of the same architecture. In comparison, SGX's microcode is CPU model-specific, so each micro-architectural revision would require a separate verification effort.

2

An Analysis of Intel's Software Guard Extensions (SGX)

Intel's Software Guard Extensions (SGX) is a set of extensions to the Intel architecture that aims to provide integrity and confidentiality guarantees to security- sensitive computation performed on a computer where the privileged software (kernel, hypervisor, etc) is potentially malicious.

This section extends the survey of Intel's SGX presented in (§ I.5). Software Guard Extensions, and analyzes SGX based on the 3 papers [McKeen et al., 2013, Anati et al., 2013, Hoekstra et al., 2013] that introduced it, on the Intel Software Developer's Manual [Int, 2015b] (which supersedes the SGX manuals [Int, 2013, 2014]), an ISCA 2015 tutorial [Int, 2015a], and two patents [McKeen et al., 2009, Johnson et al., 2010]. We use the papers, reference manuals, and tutorial as primary data sources, and only draw on the patents to fill in missing information.[1]

This section discusses the implementation of SGX (including a series of intelligent guesses of important but undocumented aspects of

[1]We note that this manuscript does not reflect the information available in two papers [Johnson et al., 2016, Gueron, 2016] that were published after [Costan and Devadas, 2016].

SGX), its security argument, and the implication of the complex Intel ecosystem on the ability of SGX to enable trusted remote computation.

2.1 SGX Implementation Overview

An under-documented and overlooked feat achieved by the SGX design is that implementing it on an Intel processor has a very low impact on the chip's hardware design. SGX's modifications to the processor's execution cores (§ I.2.9.4) are either very small or completely nonexistent. The CPU's uncore (§ I.2.9.3, § I.2.11.3) receives a new module, the Memory Encryption Engine, which appears to be fairly self-contained.

The bulk of the SGX implementation is relegated to the processor's microcode (§ I.2.14), which supports a much higher development speed than the chip's electrical circuitry.

2.1.1 Execution Core Modifications

At a minimum, the SGX design requires a very small modification to the processor's execution cores (§ I.2.9.4), in the Page Miss Handler (PMH, § I.2.11.5).

The PMH resolves TLB misses, and consists of a fast path that relies on an FSM page walker, and a microcode assist fallback that handles the edge cases (§ I.2.14.3). The bulk of SGX's memory access checks, which are discussed in § 2.2, can be implemented in the microcode assist.

The only modification to the PMH hardware that is absolutely necessary to implement SGX is developing an ability to trigger the microcode assist for all address translations when a logical processor (§ I.2.9.4) is in enclave mode (§ I.5.4), or when the physical address produced by the page walker FSM matches the Processor Reserved Memory (PRM, § I.5.1) range.

The PRM range is configured by the PRM Range Registers (§ I.5.1), which have exactly the same semantics as the Memory Type Range Registers (MTRRs, § I.2.11.4) used to configure a variable memory range. The page walker FSM in the PMH is already configured to issue a microcode assist when the page tables are in uncacheable

memory (§ I.2.11.4). Therefore, the PRMRR can be represented as an extra MTRR pair.

2.1.2 Uncore Modifications

The Intel's *Software Development Manual* (SDM) [Int, 2015b] states that DMA transactions (§ I.2.9.1) that target the PRM range are aborted by the processor. The SGX patents disclose that the PRMRR protection against unauthorized DMA is implemented by having the SGX microcode set up entries in the Source Address Decoder (SAD) in the uncore CBoxes and in the Target Address Decoder (TAD) in the integrated Memory Controller (MC).

§ I.2.11.3 mentions that Intel's Trusted Execution Technology (TXT) [Grawrock, 2009] already takes advantage of the integrated MC to protect a DRAM range from DMA. It is highly likely that the SGX implementation reuses the mechanisms brought by TXT, and only requires the extension of the SADs and TADs by one entry.

SGX's major hardware modification is the Memory Encryption Engine (MEE) that is added to the processor's uncore (§ I.2.9.3, § I.2.11.3) to protect SGX's Enclave Page Cache (EPC, § I.5.1.1) against physical attacks.

The MEE was first briefly described in the ISCA 2015 SGX tutorial [Int, 2015a]. According to the information presented there, the MEE roughly follows the approach introduced by Aegis [Suh et al., 2003] [Suh et al., 2005], which relies on a variation of Merkle trees to provide the EPC with confidentiality, integrity, and freshness guarantees (§ I.3.1). Unlike Aegis, the MEE uses non-standard cryptographic primitives that include a slightly modified AES operating mode (§ I.3.1.2) and a Carter-Wegman [Carter and Wegman, 1977, Wegman and Carter, 1981] MAC (§ I.3.1.3) construction. The MEE was further described in [Gueron, 2016].

Both the ISCA SGX tutorial and the patents state that the MEE is connected to to the Memory Controller (MC) integrated in the CPU's uncore. However, all sources are completely silent on further implementation details. The MEE overview slide states that "the Memory Controller detects [the] address belongs to the MEE region, and

routes transaction to MEE", which suggests that the MEE is fairly self-contained and has a narrow interface to the rest of the MC.

Intel's SGX patents use the name Crypto Memory Aperture (CMA) to refer to the MEE. The CMA description matches the MEE and PRM concepts, as follows. According to the patents, the CMA is used to securely store the EPC, relies on crypto controllers in the MC, and loses its keys during deep sleep. These details align perfectly with the SDM's statements regarding the MEE and PRM.

The Intel patents also disclose that the EPCM (§ I.5.1.2) and other structures used by the SGX implementation are also stored in the PRM. This rules out the possibility that the EPCM requires on-chip memory resembling the last-level cache (§ I.2.11, § I.2.11.3).

Last, the SGX patents shine a bit of light on an area that the official Intel documentation is completely silent about, namely the implementation concerns brought by computer systems with multiple processor chips. The patents state that the MEE also protects the Quick-Path Interconnect (QPI, § I.2.9.1) traffic using link-layer encryption.

2.1.3 Microcode Modifications

According to the SGX patents, all SGX instructions are implemented in microcode. This can also be deduced by reading the SDM's pseudocode for all the instructions, and realizing that it is highly unlikely that any SGX instruction can be implemented in 4 or fewer micro-ops (§ I.2.10), which is the most that can be handled by the simple decoders used in the hardware fast paths (S I.2.14.1).

The Asynchronous Enclave Exit (AEX, § I.5.4.3) behavior is also implemented in microcode. § I.2.14.2 draws on an assortment of Intel patents to conclude that hardware exceptions (§ I.2.8.2), including both faults and interrupts, trigger microcode events (§ I.2.14.2). It follows that the SGX implementation can implement AEX by modifying the hardware exception handlers in the microcode.

The SGX initialization sequence is also implemented in microcode. SGX is initialized in two phases. First, it is very likely that the boot sequence in microcode (§ I.2.14.4) was modified to initialize the registers associated with the SGX microcode. The ISCA SGX tutorial

states that the MEE' keys are initialized during the boot process. Second, SGX instructions are enabled by setting a bit in a Model-Specific Register (MSR, § I.2.4). This second phase involves enabling the MEE and configuring the SAD and TAD to protect the PRM range. Both tasks are amenable to a microcode implementation.

The SGX description in the SDM implies that the SGX implementation uses a significant number of new registers, which are only exposed to microcode. However, the SGX patents reveal that most of these registers are actually stored in DRAM.

For example, the patents state that each TCS (§ I.5.2.4) has two fields that receive the values of the DR7 and IA32_DEBUGCTL registers when the processor enters enclave mode (§ I.5.4.1), and are used to restore the original register values during enclave exit (§ I.5.4.2). The SDM documents these fields as "internal CREGs" (CR_SAVE_DR7 and CR_SAVE_DEBUGCTL), which are stated to be "hardware specific registers".

The SGX patents document a small subset of the CREGs described in the SDM, summarized in Table 2.1, as microcode registers. While in general we trust official documentation over patents, in this case we use the CREG descriptions provided by the patents, because they appear to be more suitable for implementation purposes.

From a cost-performance standpoint, the cost of register memory only seems to be justified for the state used by the PMH to implement SGX's memory access checks, which will be discussed in § 2.2. The other pieces of state listed as CREGs are accessed so infrequently that storing them in dedicated SRAM would make very little sense.

The SGX patents state that SGX requires very few hardware changes, and most of the implementation is in microcode, as a positive fact. We therefore suspect that minimizing hardware changes was a high priority in the SGX design, and that any SGX modification proposals need to be aware of this priority.

Table 2.1: The CREGs implemented by SGX.

SDM Name	**Bits**	**Scope**	**Description**
CSR_SGX_OWNEREPOCH	128	CPU Chip Package	Used by `EGETKEY` (§ I.5.7.5)
CR_ENCLAVE_MODE	1	Logical Processor	1 when executing code inside an enclave
CR_ACTIVE_SECS	16	Logical Processor	The index of the EPC page storing the current enclave's SECS
CR_TCS_LA	64	Logical Processor	The virtual address of the TCS (§ I.5.2.4) used to enter (§ I.5.4.1) the current enclave
CR_TCS_PH	16	Logical Processor	The index of the EPC page storing the TCS used to enter the current enclave
CR_XSAVE_PAGE_0	16	Logical Processor	The index of the EPC page storing the first page of the current SSA (§ I.5.2.5)

2.2 SGX Memory Access Protection

SGX guarantees that the software inside an enclave is isolated from all other software, including the software executing in other enclaves. This isolation guarantee is at the core of SGX's security model.

It is tempting to assume that the main protection mechanism in SGX is the Memory Encryption Engine (MEE) described in § 2.1.2, as it encrypts and MACs the DRAM's contents. However, the MEE sits in the processor's memory controller, which is at the edge of the on-chip memory hierarchy, below the caches (§ I.2.11). Therefore, the MEE cannot protect an enclave's memory from software attacks.

The root of SGX's protections against software attacks is a series of memory access checks which prevents the currently running software from accessing memory that does not belong to it. Specifically, non-enclave software is only allowed to access memory outside the PRM range, while the code inside an enclave is allowed to access non-PRM memory, and the EPC pages owned by the enclave.

Although it is believed [Evtyushkin et al., 2014] that SGX's access checks are performed on every memory access check, Intel's patents disclose that the checks are performed in the Page Miss Handler (PMH, § I.2.11.5), which only handles TLB misses.

2.2.1 Functional Description

The intuition behind SGX's memory access protections can be built by considering what it would take to implement the same protections in a trusted operating system or hypervisor, solely by using the page tables that direct the CPU's address translation feature (§ I.2.5).

The hypothetical trusted software proposed above can implement enclave entry (§ I.5.4.1) as a system call (§ I.2.8.1) that creates page table entries mapping the enclave's memory. Enclave exit (§ I.5.4.2) can be a symmetric system call that removes the page table entries created during enclave entry. When modifying the page tables, the system software has to consider TLB coherence issues (§ I.2.11.5) and perform TLB shootdowns when appropriate.

SGX leaves page table management under the system software's control, but it cannot trust the software to set up the page tables in any particular way. Therefore, the hypothetical design described above cannot be used by SGX as-is. Instead, at a conceptual level, the SGX implementation approximates the effect of having the page tables set up correctly by inspecting every address translation that comes out of the Page Miss Handler (PMH, § I.2.11.5). The address translations that do not obey SGX's access control restrictions are rejected before they reach the TLBs.

SGX's approach relies on the fact that software always references memory using virtual addresses, so all micro-ops (§ I.2.10) that reach the memory execution units (§ I.2.10.1) use virtual addresses that must be resolved using the TLBs before the actual memory accesses are carried out. By contrast, the processor's microcode (§ I.2.14) has the ability to issue physical memory accesses, which bypass the TLBs. Conveniently, SGX instructions are implemented in microcode (§ 2.1.3), so they can bypass the TLBs and access memory that is off limits to software, such as the EPC page holding an enclave's SECS (§ I.5.1.3).

The SGX address translation checks use the information in the Enclave Page Cache Map (EPCM, § I.5.1.2), which is effectively an inverted page table that covers the entire EPC. This means that each EPC page is accounted for by an EPCM entry, using the structure is summarized in Table 2.2, with the PT (page type) field enumerated by Table 2.3. The EPCM fields were described in detail in § I.5.1.2, § I.5.2.3, § I.5.2.4, § I.5.5.1, and § I.5.5.2.

Conceptually, SGX adds the access control logic illustrated in Figure 2.1 to the PMH. SGX's security checks are performed after the page table attributes-based checks (§ I.2.5.3) defined by the Intel architecture. It follows that SGX's access control logic has access to the physical address produced by the page walker FSM.

SGX's security checks depend on whether the logical processor (§ I.2.9.4) is in enclave mode (§ I.5.4) or not. While the processor is outside enclave mode, the PMH allows any address translation that does not target the PRM range (§ I.5.1). When the processor is inside

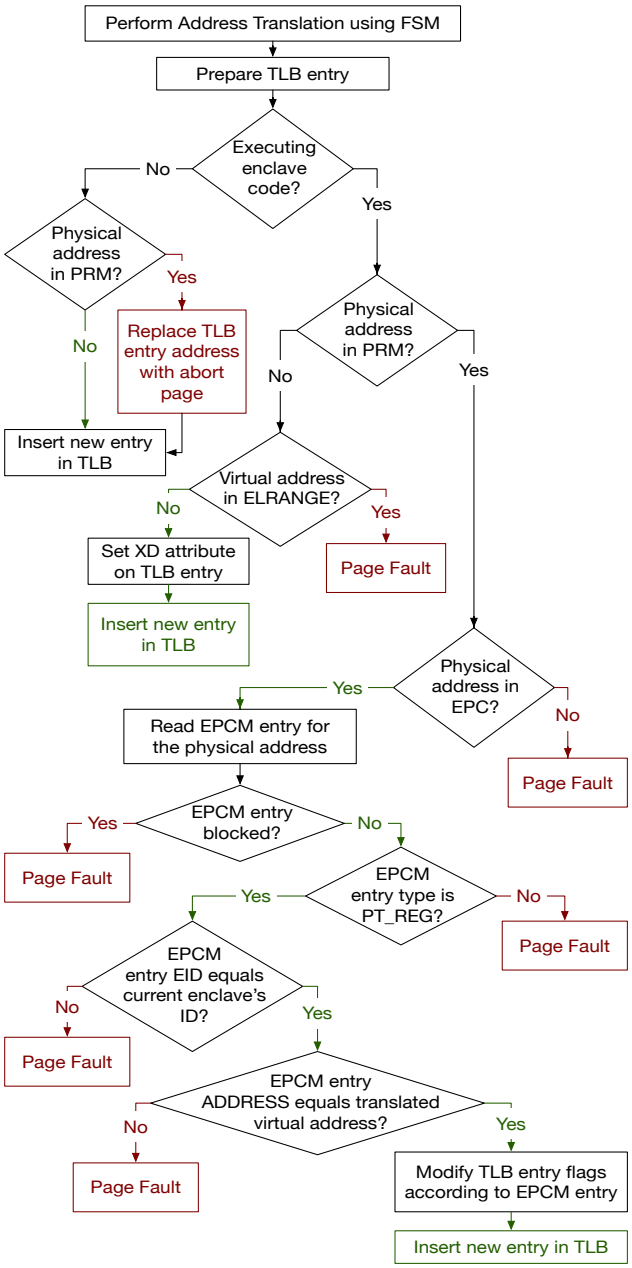

Figure 2.1: SGX adds a few security checks to the PMH. The checks ensure that all TLB entries created by the address translation unit meet SGX's memory access restrictions.

Table 2.2: The fields in an EPCM entry.

Field	Bits	Description
VALID	1	0 for un-allocated EPC pages
BLOCKED	1	page is being evicted
R	1	enclave code can read
W	1	enclave code can write
X	1	enclave code can execute
PT	8	page type (Table 2.3)
ADDRESS	48	the virtual address used to access this page
ENCLAVESECS		the EPC slot number for the SECS of the enclave owning the page

Table 2.3: Values of the PT (page type) field in an EPCM entry.

Type	Allocated by	Contents
PT_REG	`EADD`	enclave code and data
PT_SECS	`ECREATE`	SECS (§ I.5.1.3)
PT_TCS	`EADD`	TCS (§ I.5.2.4)
PT_VA	`EPA`	VA (§ I.5.5.2)

enclave mode, the PMH performs the checks described below, which provide the security guarantees described in § I.5.2.3.

First, virtual addresses inside the enclave's virtual memory range (ELRANGE, § I.5.2.1) must always translate into physical addresses inside the EPC. This way, an enclave is assured that all code and data stored in ELRANGE is covered by SGX's confidentiality, integrity, and freshness guarantees. Since the memory outside ELRANGE does not enjoy these guarantees, the SGX design disallows having enclave code outside ELRANGE. This is most likely accomplished by setting the disable execution (XD, § I.2.5.3) attribute on the TLB entry.

Second, an EPC page must only be accessed by the code of the enclave who owns the page. For the purpose of this check, each enclave is identified by the index of the EPC page that stores the enclave's SECS (§ I.5.1.3). The current enclave's identifier is stored

in the CR_ACTIVE_SECS microcode register during enclave entry. This register is compared against the enclave identifier stored in the EPCM entry corresponding to the EPC page targeted by the address translation.

Third, some EPC pages cannot be accessed by software. Pages that hold SGX internal structures, such as a SECS, a TCS (§ I.5.2.4), or a VA (§ I.5.5.2) must only be accessed by SGX's microcode, which uses physical addresses and bypasses the address translation unit, including the PMH. Therefore, the PMH rejects address translations targeting these pages.

Blocked (§ I.5.5.1) EPC pages are in the process of being evicted (§ I.5.5), so the PMH must not create new TLB entries targeting them.

Next, an enclave's EPC pages must always be accessed using the virtual addresses associated with them when they were allocated to the enclave. Regular EPC pages, which can be accessed by software, are allocated to enclaves using the `EADD` (§ I.5.3.2) instruction, which reads in the page's address in the enclave's virtual address space. This address is stored in the LINADDR field in the corresponding EPCM entry. Therefore it is sufficient for the PMH to is to ensure that LINADDR in the address translation's target EPCM entry equals the virtual address that caused the TLB miss which invoked the PMH.

At this point, the PMH's security checks have completed, and the address translation result will definitely be added to the TLB. Before that happens, however, the SGX extensions to the PMH apply the access restrictions in the EPCM entry for the page to the address translation result. While the public SGX documentation we found did not describe this process, there is a straightforward implementation that fulfills SGX's security requirements. Specifically, the TLB entry bits P, W, and XD can be AND-ed with the EPCM entry bits R, W, and X.

2.2.2 EPCM Entry Representation

Most EPCM entry fields have obvious representations. The exception is the LINADDR and ENCLAVESECS fields, described below. These

representations explain SGX's seemingly arbitrary limit on the size of an enclave's virtual address range (ELRANGE).

The SGX patents disclose that the LINADDR field in an EPCM entry stores the virtual page number (VPN, § I.2.5.1) of the corresponding EPC page's expected virtual address, relative to the ELRANGE base of the enclave that owns the page.

The representation described above reduces the number of bits needed to store LINADDR, assuming that the maximum ELRANGE size is significantly smaller than the virtual address size supported by the CPU. This desire to save EPCM entry bits is the most likely motivation for specifying a processor model-specific ELRANGE size, which is reported by the `CPUID` instruction.

The SDM states that the ENCLAVESECS field of an EPCM entry corresponding to an EPC page indicates the SECS of the enclave that owns the page. Intel's patents reveal that the SECS address in ENCLAVESECS is represented as a physical page number (PPN, § I.2.5.1) relative to the start of the EPC. Effectively, this relative PPN is the 0-based EPC page index.

The EPC page index representation saves bits in the EPCM entry, assuming that the EPCM size is significantly smaller than the physical address space supported by the CPU. The ISCA 2015 SGX tutorial slides mention an EPC size of 96MB, which is significantly smaller than the physical addressable space on today's typical processors, which is 2^{36} - 2^{40} bytes.

2.2.3 PMH Hardware Modifications

The SDM describes the memory access checks performed after SGX is enabled, but does not provide any insight into their implementation. Intel's patents hint at three possible implementations that make different cost-performance tradeoffs. This section summarizes the three approaches and argues in favor of the implementation that requires the fewest hardware modifications to the PMH.

All implementations of SGX's security checks entail adding a pair of memory type range registers (MTRRs, § I.2.11.4) to the PMH. These registers are named the *Secure Enclave Range Registers* (SERR)

in Intel's patents. Enabling SGX on a logical processor initializes the SERR to the values of the Protected Memory Range Registers (PMRR, § I.5.1).

Furthermore, all implementations have the same behavior when a logical processor is outside enclave mode. The memory type range described by the SERR is enabled, causing a microcode assist to trigger for every address translation that resolves inside the PRM. SGX's implementation uses the microcode assist to replace the address translation result with an address that causes memory access transactions to be aborted.

The three implementations differ in their behavior when the processor enters enclave mode (§ I.5.4) and starts executing enclave code.

The alternative that requires the least amount of hardware changes sets up the PMH to trigger a microcode assist for every address translation. This can be done by setting the SERR to cover the whole of physical memory (e.g., by setting the mask to zero, and the base to a non-zero value). In this approach, the microcode assist implements all enclave mode security checks illustrated in Figure 2.1.

A speedier alternative adds a pair of registers to the PMH that represents the current enclave's ELRANGE and modifies the PMH so that, in addition to checking physical addresses against the SERR, it also checks the virtual addresses going into address translations against ELRANGE. When either check is true, the PMH invokes the microcode assist used by SGX to implement its memory access checks. Assuming the ELRANGE registers use the same base / mask representation as variable MTRRs, enclave exits can clear ELRANGE by zeroing both the base and the mask. This approach uses the same microcode assist implementation, minus the ELRANGE check that moves into the PMH hardware.

The second alternative described above has the benefit that the microcode assist is not invoked for enclave mode accesses outside ELRANGE. However, § I.5.2.1 argues that an enclave should treat all virtual memory addresses outside ELRANGE as untrusted storage, and only use that memory to communicate with software outside the enclave. Taking this into consideration, well-designed enclaves would

spend relatively little time performing memory accesses outside ELRANGE. Therefore, this second alternative is unlikely to obtain performance gains that are worth its cost.

The last and most performant alternative would entail implementing the access checks shown in Figure 2.1 in hardware. Similarly to the address translation FSM, the hardware would only invoke a microcode assist when a security check fails and a Page Fault needs to be handled.

The high-performance implementation described above avoids the cost of microcode assists for all TLB misses, assuming well-behaved system software. In this association, a microcode assist results in a Page Fault, which triggers an Asynchronous Enclave Exit (AEX, § I.5.4.3). The cost of the AEX dominates the performance overhead of the microcode assist.

While this last implementation looks attractive, one needs to realize that TLB misses occur quite infrequently, so a large improvement in the TLB miss speed translates into a much less impressive improvement in overall enclave code execution performance. Taking this into consideration, it seems unwise to commit to extensive hardware modifications in the PMH before SGX gains adoption.

2.3 SGX Security Check Correctness

In § 2.2.1, we argued that SGX's security guarantees can be obtained by modifying the Page Miss Handler (PMH, § I.2.11.5) to block undesirable address translations from reaching the TLB. This section builds on the result above and outlines a correctness proof for SGX's memory access protection.

Specifically, we outline a proof for the following invariant. **At all times, the TLB entries in every logical processor are be consistent with the SGX security policy.** By the argument in § 2.2.1, the invariant translates into an assurance that all memory accesses performed by software obey SGX's security model. The high-level proof structure is presented because it helps understand how the SGX security checks come together. By contrast, a detailed proof

would be incredibly tedious, and would do very little to boost the reader's understanding of SGX.

2.3.1 Top-Level Invariant Breakdown

We first break down the above invariant into specific cases based on whether a logical processor (LP) is executing enclave code or not, and on whether the TLB entries translate virtual addresses in the current enclave's ELRANGE (§ I.5.2.1). When the processor is outside enclave mode, ELRANGE can be considered to be empty. This reasoning yields the three cases outlined below.

1. At all times when an LP is outside enclave mode, its TLB may only contain physical addresses belonging to DRAM pages outside the PRM.

2. At all times when an LP is inside enclave mode, the TLB entries for virtual addresses outside the current enclave's ELRANGE must contain physical addresses belonging to DRAM pages outside the PRM.

3. At all times when an LP is in enclave mode, the TLB entries for virtual addresses inside the current enclave's ELRANGE must match the virtual memory layout specified by the enclave author.

The first two invariant cases can be easily proven independently for each LP, by induction over the sequence of instructions executed by the LP. For simplicity, the reader can assume that instructions are executed in program mode. While the assumption is not true on processors with out-of-order execution (§ I.2.10), the arguments presented here also hold when the executed instruction sequence is considered in retirement order, for reasons that will be described below.

An LP will only transition between enclave mode and non-enclave mode at a few well-defined points, which are `EENTER` (§ I.5.4.1), `ERESUME` (§ I.5.4.4), `EEXIT` (§ I.5.4.2), and Asynchronous Enclave Exits (AEX, § I.5.4.3). According to the SDM, all transition points flush the TLBs and the out-of-order execution pipeline. In other words, the

TLBs are guaranteed to be empty after every transition between enclave mode and non-enclave mode, so we can consider all these transitions to be trivial base cases for our induction proofs.

While SGX initialization is not thoroughly discussed, the SDM mentions that loading some Model-Specific Registers (MSRs, § I.2.4) triggers TLB flushes, and that system software should flush TLBs when modifying Memory Type Range Registers (MTRRs, § I.2.11.4). Given that the space of possible SGX implementations described in § 2.2.3 entails adding a MTRR, it is safe to assume that enabling SGX mode also results in a TLB flush and out-of-order pipeline flush, and can be used by our induction proof as well.

The base cases in the induction proofs are serialization points for out-of-order execution, as the pipeline is flushed during both enclave mode transitions and SGX initialization. This makes the proofs below hold when the program order instruction sequence is replaced with the retirement order sequence.

The first invariant case holds because while the LP is outside enclave mode, the SGX security checks added to the PMH (§ 2.2.1, Figure 2.1) reject any address translation that would point into the PRM before it reaches the TLBs. A key observation for proving the induction step of this invariant case is that the PRM never changes after SGX is enabled on an LP.

The second invariant case can be proved using a similar argument. While an LP is executing an enclave's code, the SGX memory access checks added to the PMH reject any address translation that resolves to a physical address inside the PRM, if the translated virtual address falls outside the current enclave's ELRANGE. The induction step for this invariant case can be proven by observing that a change in an LP's current ELRANGE is always accompanied by a TLB flush, which results in an empty TLB that trivially satisfies the invariant. This follows from the constraint that an enclave's ELRANGE never changes after it is established, and from the observation that the LP's current enclave can only be changed by an enclave entry, which must be preceded by an enclave exit, which triggers a TLB flush.

The third invariant case is best handled by recognizing that the Enclave Page Cache Map (EPCM, § I.5.1.2) is an intermediate representation for the virtual memory layout specified by the enclave authors. This suggests breaking down the case into smaller sub-invariants centered around the EPCM, which will be proven in the subsections below.

1. At all times, each EPCM entry for a page that is allocated to an enclave matches the virtual memory layout desired by the enclave's author.

2. Assuming that the EPCM contents is constant, at all times when an LP is in enclave mode, the TLB entries for virtual addresses inside the current enclave's ELRANGE must match EPCM entries that belong to the enclave.

3. An EPCM entry is only modified when there is no mapping for it in any LP's TLB.

The second and third invariant combined prove that all TLBs in an SGX-enabled computer always reflect the contents of the EPCM, as the third invariant essentially covers the gaps in the second invariant. This result, in combination with the first invariant, shows that the EPCM is a bridge between the memory layout specifications of the enclave authors and the TLB entries that regulate what memory can be accessed by software executing on the LPs. When further combined with the reasoning in § 2.2.1, the whole proof outlined here results in an end-to-end argument for the correctness of SGX's memory protection scheme.

2.3.2 EPCM Entries Reflect Enclave Author Design

This subsection outlines the proof for the following invariant. **At all times, each EPCM entry for a page that is allocated to an enclave matches the virtual memory layout desired by the enclave's author.**

A key observation, backed by the SDM pseudocode for SGX instructions: all instructions that modify the EPCM pages allocated to an enclave are synchronized using a lock in the enclave's SECS. This

entails the existence of a time ordering of the EPCM modifications associated with an enclave. We prove the invariant stated above using a proof by induction over this sequence of EPCM modifications.

EPCM entries allocated to an enclave are created by instructions that can only be issued before the enclave is initialized, specifically `ECREATE` (§ I.5.3.1) and `EADD` (§ I.5.3.2). The contents of the EPCM entries created by these instructions contributes to the enclave's measurement (§ I.5.6), together with the initial data loaded into the corresponding EPC pages.

§ I.3.3.3 argues that we can assume that enclaves with incorrect measurements do not exist, as they will be rejected by software attestation. Therefore, we can assume that the attributes used to initialize EPCM pages match the enclave authors' memory layout specifications.

EPCM entries can be evicted to untrusted DRAM, together with their corresponding EPC pages, by the `EWB` (§ I.5.5.4) instruction. The `ELDU` / `ELDB` (§ I.5.5) instructions reload evicted page contents and metadata back into the EPC and EPCM. By induction, we can assume that an EPCM entry matches the enclave author's specification when it is evicted. Therefore, if we can prove that the EPCM entry that is reloaded from DRAM is equivalent to the entry that was evicted, we can conclude that the reloaded entry matches the author's specification.

A detailed analysis of the cryptographic primitives used by the SGX design to protect the evicted EPC page contents and its associated metadata is outside the scope of this work. Summarizing the description in § I.5.5, the contents of evicted pages is encrypted using AES-GMAC (§ I.3.1.3), which is an authenticated encryption mechanism. The MAC tag produced by AES-GMAC covers the EPCM metadata as well as the page data, and includes a 64-bit version that is stored in a version tree whose nodes are Version Array (VA, (§ I.5.5.2) pages.

Assuming no cryptographic weaknesses, SGX's scheme does appear to guarantee the confidentiality, integrity, and freshness of the EPC page contents and associated metadata while it is evicted to untrusted memory. It follows that `EWB` will only reload an EPCM entry if the contents is equivalent to the contents of an evicted entry.

The equivalence notion invoked here is slightly different from perfect equality, in order to account for the allowable operation of evicting an EPC page and its associated EPCM entry, and then reloading the page contents to a different EPC page and a different EPCM entry, as illustrated in Figure I.5.10. Loading the contents of an EPC page at a different physical address than it had before does not break the virtual memory abstraction, as long as the contents is mapped at the same virtual address (the LINEARADDRESS EPCM field), and has the same access control attributes (R, W, X, PT EPCM fields) as it had when it was evicted.

The rest of this section enumerates the address translation attacks prevented by the MAC verification that occurs in `ELDU` / `ELDB`. This is intended to help the reader develop some intuition for the reasoning behind using the page data and all EPCM fields to compute and verify the MAC tag.

The most obvious attack is prevented by having the MAC cover the contents of the evicted EPC page, so the untrusted OS cannot modify the data in the page while it is stored in untrusted DRAM. The MAC also covers the metadata that makes up the EPCM entry, which prevents the more subtle attacks described below.

The enclave ID (EID) field is covered by the MAC tag, so the OS cannot evict an EPC page belonging to one enclave, and assign the page to a different enclave when it is loaded back into the EPC. If EID was not covered by authenticity guarantees, a malicious OS could read any enclave's data by evicting an EPC page belonging to the victim enclave, and loading it into a malicious enclave that would copy the page's contents to untrusted DRAM.

The virtual address (LINADDR) field is covered by the MAC tag, so the OS cannot modify the virtual memory layout of an enclave by evicting an EPC page and specifying a different LINADDR when loading it back. If LINADDR was not covered by authenticity guarantees, a malicious OS could perform the exact attack shown in Figure I.3.24 and described in § I.3.7.3.

The page access permission flags (R, W, X) are also covered by the MAC tag. This prevents the OS from changing the access permis-

sion bits in a page's EPCM entry by evicting the page and loading it back in. If the permission flags were not covered by authenticity guarantees, the OS could use the ability to change EPCM access permissions to facilitate exploiting vulnerabilities in enclave code. For example, exploiting a stack overflow vulnerability is generally easier if OS can make the stack pages executable.

The nonce stored in the VA slot is also covered by the MAC. This prevents the OS from mounting a replay attack that reverts the contents of an EPC page to an older version. If the nonce would not be covered by integrity guarantees, the OS could evict the target EPC page at different times t_1 and t_2 in the enclave's life, and then provide the `EWB` outputs at t_1 to the `ELDU` / `ELDB` instruction. Without the MAC verification, this attack would successfully revert the contents of the EPC page to its version at t_1.

While replay attacks look relatively benign, they can be quite devastating when used to facilitate double spending.

2.3.3 TLB Entries for ELRANGE Reflect EPCM Contents

This subsection sketches a proof for the following invariant. **At all times when an LP is in enclave mode, the TLB entries for virtual addresses inside the current enclave's ELRANGE must match EPCM entries that belong to the enclave.** The argument makes the assumption that **the EPCM contents is constant**, which will be justified in the following subsection.

The invariant can be proven by induction over the sequence of TLB insertions that occur in the LP. This sequence is well-defined because an LP has a single PMH, so the address translation requests triggered by TLB misses must be serialized to be processed by the PMH.

The proof's induction step depends on the fact that the TLB on hyper-threaded cores (§ I.2.9.4) is dynamically partitioned between the two LPs that share the core, and no TLB entry is shared between the LPs. This allows our proof to consider the TLB insertions associated with one LP independently from the other LP's insertions, which means we don't have to worry about the state (e.g., enclave mode) of the other LP on the core.

The proof is further simplified by observing that when an LP exits enclave mode, both its TLB and its out-of-order instruction pipeline are flushed. Therefore, the enclave mode and current enclave register values used by address translations are guaranteed to match the values obtained by performing the translations in program order.

Having eliminated the complexities associated with hyper-threaded (§ I.2.9.4) out-of-order (§ I.2.10) execution cores, it is easy to see that the security checks outlined in Figure 2.1 and § 2.2.1 ensure that TLB entries that target EPC pages are guaranteed to reflect the constraints in the corresponding EPCM entries.

Last, the SGX access checks implemented in the PMH reject any address translation for a virtual address in ELRANGE that does not resolve to an EPC page. It follows that memory addresses inside ELRANGE can only map to EPC pages which, by the argument above, must follow the constraints of the corresponding EPCM entries.

2.3.4 EPCM Entries are Not In TLBs When Modified

In this subsection, we outline a proof that **an EPCM entry is only modified when there is no mapping for it in any LP's TLB.**. This proof analyzes each of the instructions that modify EPCM entries.

For the purposes of this proof, we consider that setting the BLOCKED attribute does not count as a modification to an EPCM entry, as it does not change the EPC page that the entry is associated with, or the memory layout specification associated with the page.

The instructions that modify EPCM entries in such a way that the resulting EPCM entries have the VALID field set to true require that the EPCM entries were invalid before they were modified. These instructions are `ECREATE` (§ I.5.3.1), `EADD` (§ I.5.3.2), `EPA` (§ I.5.5.2), and `ELDU` / `ELDB` (§ I.5.5). The EPCM entry targeted by any these instructions must have had its VALID field set to false, so the invariant proved in the previous subsection implies that the EPCM entry had no TLB entry associated with it.

Conversely, the instructions that modify EPCM entries and result in entries whose VALID field is false start out with valid entries. These instructions are `EREMOVE` (§ I.5.3.4) and `EWB` (§ I.5.5.4).

The EPCM entries associated with EPC pages that store Version Arrays (VA, § I.5.5.2) represent a special case for both instructions mentioned above, as these pages are not associated with any enclave. As these pages can only be accessed by the microcode used to implement SGX, they never have TLB entries representing them. Therefore, both `EREMOVE` and `EWB` can invalidate EPCM entries for VA pages without additional checks.

`EREMOVE` only invalidates an EPCM entry associated with an enclave when there is no LP executing in enclave mode using a TCS associated with the same enclave. An EPCM entry can only result in TLB translations when an LP is executing code from the entry's enclave, and the TLB translations are flushed when the LP exits enclave mode. Therefore, when `EREMOVE` invalidates an EPCM entry, any associated TLB entry is guaranteed to have been flushed.

`EWB`'s correctness argument is more complex, as it relies on the `EBLOCK` / `ETRACK` sequence described in § I.5.5.1 to ensure that any TLB entry that may have been created for an EPCM entry is flushed before the EPCM entry is invalidated.

Unfortunately, the SDM pseudocode for the instructions mentioned above leaves out the algorithm used to verify that the relevant TLB entries have been flushed. Thus, we must base our proof on the assumption that the SGX implementation produced by Intel's engineers matches the claims in the SDM. In § 2.4, we propose a method for ensuring that `EWB` will only succeed when all LPs executing an enclave's code at the time when `ETRACK` is called have exited enclave mode at least once between the `ETRACK` call and the `EWB` call. Having proven the existence of a correct algorithm by construction, we can only hope that the SGX implementation uses our algorithm, or a better algorithm that is still correct.

2.4 Tracking TLB Flushes

This section proposes a straightforward method that the SGX implementation can use to verify that the system software plays its part correctly in the EPC page eviction (§ I.5.5) process. Our method

meets the SDM's specification for `EBLOCK` (§ I.5.5.1), `ETRACK` (§ I.5.5.1) and `EWB` (§ I.5.5.4).

The motivation behind this section is that, at least at the time of this writing, there is no official SGX documentation that contains a description of the mechanism used by `EWB` to ensure that all Logical Processors (LPs, § I.2.9.4) running an enclave's code exit enclave mode (§ I.5.4) between an `ETRACK` invocation and a `EWB` invocation. Knowing that there exists a correct mechanism that has the same interface as the SGX instructions described in the SDM gives us a reason to hope that the SGX implementation is also correct.

Our method relies on the fact that an enclave's SECS (§ I.5.1.3) is not accessible by software, and is already used to store information used by the SGX microcode implementation (§ 2.1.3). We store the following fields in the SECS. *tracking* and *done-tracking* are Boolean variables. *tracked-threads* and *active-threads* are non-negative integers that start at zero and must store numbers up to the number of LPs in the computer. *lp-mask* is an array of Boolean flags that has one member per LP in the computer. The fields are initialized as shown in Figure 2.2.

```
ECREATE(SECS)
    ▷ Initialize the SECS state used for tracking.
1   SECS . tracking ← FALSE
2   SECS . done-tracking ← FALSE
3   SECS . active-threads ← 0
4   SECS . tracked-threads ← 0
5   SECS . lp-mask ← 0
```

Figure 2.2: The algorithm used to initialize the SECS fields used by the TLB flush tracking method presented in this section.

The *active-threads* SECS field tracks the number of LPs that are currently executing the code of the enclave who owns the SECS. The field is atomically incremented by `EENTER` (§ I.5.4.1) and `ERESUME` (§ I.5.4.4) and is atomically decremented by `EEXIT` (§ I.5.4.2) and Asynchronous Enclave Exits (AEXs, § I.5.4.3). Asides from helping

track TLB flushes, this field can also be used by `EREMOVE` (§ I.5.3.4) to decide when it is safe to free an EPC page that belongs to an enclave.

As specified in the SDM, `ETRACK` activates TLB flush tracking for an enclave. In our method, this is accomplished by setting the *tracking* field to TRUE and the *done-tracking* field to FALSE.

When tracking is enabled, *tracked-threads* is the number of LPs that were executing the enclave's code when the `ETRACK` instruction was issued, and have not yet exited enclave mode. Therefore, executing `ETRACK` atomically reads *active-threads* and writes the result into *tracked-threads*. Also, *lp-mask* keeps track of the LPs that have exited the current enclave after the `ETRACK` instruction was issued. Therefore, the `ETRACK` implementation atomically zeroes *lp-mask*. The full `ETRACK` algorithm is listed in Figure 2.3.

```
ETRACK(SECS)
   ▷ Abort if tracking is already active.
1  if SECS . tracking = TRUE
2     then return SGX-PREV-TRK-INCMPL
   ▷ Activate TLB flush tracking.
3  SECS . tracking ← TRUE
4  SECS . done-tracking ← FALSE
5  SECS . tracked-threads ←
             ATOMIC-READ(SECS . active-threads)
6  for i ← 0 to MAX-LP-ID
7        do ATOMIC-CLEAR(SECS . lp-mask[i])
```

Figure 2.3: The algorithm used by `ETRACK` to activate TLB flush tracking.

When an LP exits an enclave that has TLB flush tracking activated, we atomically test and set the current LP's flag in *lp-mask*. If the flag was not previously set, it means that an LP that was executing the enclave's code when `ETRACK` was invoked just exited enclave mode for the first time, and we atomically decrement *tracked-threads* to reflect

this fact. In other words, *lp-mask* prevents us from double-counting an LP when it exits the same enclave while TLB flush tracking is active.

Once *active-threads* reaches zero, we are assured that all LPs running the enclave's code when `ETRACK` was issued have exited enclave mode at least once, and can set the *done-tracking* flag. Figure 2.4 enumerates the steps taken during an enclave exit.

```
ENCLAVE-EXIT(SECS)
    ▷ Track an enclave exit.
1   ATOMIC-DECREMENT(SECS . active-threads)
2   if ATOMIC-TEST-AND-SET(
               SECS . lp-mask[LP-ID])
3      then ATOMIC-DECREMENT(
                        SECS . tracked-threads)
4               if SECS . tracked-threads = 0
5                  then SECS . done-tracking ← TRUE
```

Figure 2.4: The algorithm that updates the TLB flush tracking state when an LP exits an enclave via `EEXIT` or AEX.

Without any compensating measure, the method above will incorrectly decrement *tracked-threads*, if the LP exiting the enclave had entered it after `ETRACK` was issued. We compensate for this with the following trick. When an LP starts executing code inside an enclave that has TLB flush tracking activated, we set its corresponding flag in *lp-mask*. This is sufficient to avoid counting the LP when it exits the enclave. Figure 2.5 lists the steps required by our method when an LP enters an enclave.

With these algorithms in place, `EWB` can simply verify that both *tracking* and *done-tracking* are TRUE. This ensures that the system software has triggered enclave exits on all LPs that were running the enclave's code when `ETRACK` was executed. Figure 2.6 lists the algorithm used by the `EWB` tracking verification step.

Last, `EBLOCK` marks the end of a TLB flush tracking cycle by clearing the *tracking* flag. This ensures that system software must go through

```
ENCLAVE-ENTER(SECS)
    ▷ Track an enclave entry.
1   ATOMIC-INCREMENT(SECS . active-threads)
2   ATOMIC-SET(SECS . lp-mask[LP-ID])
```

Figure 2.5: The algorithm that updates the TLB flush tracking state when an LP enters an enclave via `EENTER` or `ERESUME`.

another cycle of `ETRACK` and enclave exits before being able to use `EWB` on the page whose BLOCKED EPCM field was just set to TRUE by `EBLOCK`. Figure 2.7 shows the details.

Our method's correctness can be easily proven by arguing that each SECS field introduced in this section has its intended value throughout enclave entries and exits.

2.5 Enclave Signature Verification

Let m be the public modulus in the enclave author's RSA key, and s be the enclave signature. Since the SGX design fixes the value of the public exponent e to 3, verifying the RSA signature amounts to computing the signed message $M = s^3 \bmod m$, checking that the value meets the PKCS v1.5 padding requirements, and comparing the 256-bit SHA-2 hash inside the message with the value obtained by hashing the relevant fields in the SIGSTRUCT supplied with the enclave.

This section describes an algorithm for computing the signed message while only using subtraction and multiplication on large non-negative integers. The algorithm admits a significantly simpler implementation than the typical RSA signature verification algorithm, by avoiding the use of long division and negative numbers. The description here is essentially the idea in [Gueron, 2011], specialized for $e = 3$.

The algorithm provided here requires the signer to compute the q_1 and q_2 values shown below. The values can be computed from the public information in the signature, so they do not leak any additional information about the private signing key. Furthermore, the algorithm

```
EWB-VERIFY(virtual-addr)
1  physical-addr ← TRANSLATE(virtual-addr)
2  epcm-slot ← EPCM-SLOT(physical-addr)
3  if EPCM[slot]. BLOCKED = FALSE
4     then return SGX-NOT-BLOCKED
5  SECS ← EPCM-ADDR(
              EPCM[slot]. ENCLAVESECS)
   ▷ Verify that the EPC page can be evicted.
6  if SECS . tracking = FALSE
7     then return SGX-NOT-TRACKED
8  if SECS . done-tracking = FALSE
9     then return SGX-NOT-TRACKED
```

Figure 2.6: The algorithm that ensures that all LPs running an enclave's code when `ETRACK` was executed have exited enclave mode at least once.

verifies the correctness of the values, so it does not open up the possibility for an attack that relies on supplying incorrect values for q_1 and q_2.

$$q_1 = \left\lfloor \frac{s^2}{m} \right\rfloor$$
$$q_2 = \left\lfloor \frac{s^3 - q_1 \times s \times m}{m} \right\rfloor$$

Due to the desirable properties mentioned above, it is very likely that the algorithm described here is used by the SGX implementation to verify the RSA signature in an enclave's SIGSTRUCT (§ I.5.7.1).

The algorithm given by Figure 2.8 computes the signed message $M = s^3 \bmod m$, while also verifying that the given values of q_1 and q_2 are correct. The latter is necessary because the SGX implementation of signature verification must handle the case where an attacker attempts to exploit the signature verification implementation by supplying invalid values for q_1 and q_2.

```
EBLOCK(virtual-addr)
1  physical-addr ← TRANSLATE(virtual-addr)
2  epcm-slot ← EPCM-SLOT(physical-addr)
3  if EPCM[slot].BLOCKED = TRUE
4     then return SGX-BLKSTATE
5  if SECS.tracking = TRUE
6     then if SECS.done-tracking = FALSE
7             then return SGX-ENTRYEPOCH-LOCKED
8          SECS.tracking ← FALSE
9  EPCM[slot].BLOCKED ← TRUE
```

Figure 2.7: The algorithm that marks the end of a TLB flushing cycle when `EBLOCK` is executed.

1. Compute $u \leftarrow s \times s$ and $v \leftarrow q_1 \times m$.
2. If $u < v$, abort. q_1 must be incorrect.
3. Compute $w \leftarrow u - v$.
4. If $w \geq m$, abort. q_1 must be incorrect.
5. Compute $x \leftarrow w \times s$ and $y \leftarrow q_2 \times m$
6. If $x < y$, abort. q_2 must be incorrect.
7. Compute $z \leftarrow x - y$.
8. If $z \geq m$, abort. q_2 must be incorrect.
9. Output z.

Figure 2.8: An RSA signature verification algorithm specialized for the case where the public exponent is 3. s is the RSA signature and m is the RSA key modulus. The algorithm uses two additional inputs, q_1 and q_2.

The rest of this section proves the correctness of the RSA signature verification.

2.5.1 Analysis of Steps 1 - 4

Steps $1-4$ in the algorithm check the correctness of q_1 and use it to compute $s^2 \bmod m$. The key observation to understanding these steps is recognizing that q_1 is the quotient of the integer division s^2/m.

Having made this observation, we can use elementary division properties to prove that the supplied value for q_1 is correct if and only if the following property holds.

$$0 \leq s^2 - q_1 \times m < m$$

We observe that the first comparison, $0 \leq s^2 - q_1 \times m$, is equivalent to $q_1 \times m \leq s^2$, which is precisely the check performed by step 2. We can also see that the second comparison, $s^2 - q_1 \times m < m$ corresponds to the condition verified by step 4. Therefore, if the algorithm passes step 4, it must be the case that the value supplied for q_1 is correct.

We can also plug s^2, q_1 and m into the integer division remainder definition to obtain the identity $s^2 \bmod m = s^2 - q_1 \times m$. However, according to the computations performed in steps 1 and 3, $w = s^2 - q_1 \times m$. Therefore, we can conclude that $w = s^2 \bmod m$.

2.5.2 Analysis of Steps 5 - 8

Similarly, steps $5-8$ in the algorithm check the correctness of q_2 and use it to compute $w \times s \bmod m$. The key observation here is that q_2 is the quotient of the integer division $(w \times s)/m$.

We can convince ourselves of the truth of this observation by using the fact that $w = s^2 \bmod m$, which was proven above, by plugging in the definition of the remainder in integer division, and by taking advantage of the distributivity of integer multiplication with respect to addition.

$$\begin{aligned}
\left\lfloor \frac{w \times s}{m} \right\rfloor &= \left\lfloor \frac{(s^2 \bmod m) \times s}{m} \right\rfloor \\
&= \left\lfloor \frac{(s^2 - \lfloor \frac{s^2}{m} \rfloor \times m) \times s}{m} \right\rfloor \\
&= \left\lfloor \frac{s^3 - \lfloor \frac{s^2}{m} \rfloor \times m \times s}{m} \right\rfloor \\
&= \left\lfloor \frac{s^3 - q_1 \times m \times s}{m} \right\rfloor \\
&= \left\lfloor \frac{s^3 - q_1 \times s \times m}{m} \right\rfloor \\
&= q_2
\end{aligned}$$

By the same argument used to analyze steps $1-4$, we use elementary division properties to prove that q_2 is correct if and only if the equation below is correct.

$$0 \leq w \times s - q_2 \times m < m$$

The equation's first comparison, $0 \leq w \times s - q_2 \times m$, is equivalent to $q_2 \times m \leq w \times s$, which corresponds to the check performed by step 6. The second comparison, $w \times s - q_2 \times m < m$, matches the condition verified by step 8. It follows that, if the algorithm passes step 8, it must be the case that the value supplied for q_2 is correct.

By plugging $w \times s$, q_2 and m into the integer division remainder definition, we obtain the identity $w \times s \bmod m = w \times s - q_2 \times m$. Trivial substitution reveals that the computations in steps 5 and 7 result in $z = w \times s - q_2 \times m$, which allows us to conclude that $z = w \times s \bmod m$.

In the analysis for steps $1-4$, we have proven that $w = s^2 \bmod m$. By substituting this into the above identity, we obtain the proof that the algorithm's output is indeed the desired signed message.

$$\begin{aligned} z &= w \times s \bmod m \\ &= (s^2 \bmod m) \times s \bmod m \\ &= s^2 \times s \bmod m \\ &= s^3 \bmod m \end{aligned}$$

2.5.3 Implementation Requirements

The main advantage of SGX's RSA signature verification is that it relies on the implementation of very few arithmetic operations on large integers. The maximum integer size that needs to be handled is twice the size of the modulus in the RSA key used to generate the signature.

Steps 1 and 5 use large integer multiplication. Steps 3 and 7 use integer subtraction. Steps 2, 4, 6, and 8 use large integer comparison. The checks in steps 2 and 6 guarantee that the results of the subtractions performed in steps 3 and 7 will be non-negative. It follows that the algorithm will never encounter negative numbers.

2.6 Key Hierarchy and Derivation

According to Intel's patents, the SGX implementation relies on a complex key derivation process rooted on global secret keys in the CPU circuitry, and on secrets embedded in the processor's eFUSEs. eFUSE information can be extracted efficiently (Chipworks quotes $50-250k for extracting the entire eFUSE contents from an Intel i5 processor), so some of the eFUSE secrets are encrypted with a master key (referred to as a "global wrapping logic key" in the patents).

The SGX patents describe two "logic keys" embedded in the CPU's circuitry, which are the same for all CPUs in a stepping, making them essentially global keys. The *global wrapping logic key* (GWK) is a 128-bit AES key, and it is used to encrypt a subset a 256-bit A.x value used to re-create the CPU's EPID key, and a 128-bit *pre-seed key 0*. The eFUSEs also contain a 128-bit *pre-seed key 1* and a 32-bit EPID group ID, which are stored in cleartext.

[Shanbhogue et al., 2015], which describes mechanisms for protecting SGX keys in the presence of hardware debuggers, states that some hardware secrets are stored in "metal tie-ups and tie-downs", while other secrets are stored in e-fuses or in *physically unclonable function* (PUF) circuits [Gassend et al., 2002]. [Brickell and Li, 2014, Gotze et al., 2014a] state that the GWK is embedded into a chip using "metal tie-ups and tie-downs", and is shared by all chips manufactured from the same mask set.

The SGX patents state that encrypting the eFUSE secrets by the logic key makes them harder to extract via hardware monitoring tools, and protects them while in transit to the CPU during the manufacturing process. This assumes that it is very expensive to obtain the global key from a CPU, by virtue of the low feature size. [Gotze et al., 2014a] expresses concerns that the GWK can be reverse-engineered.

[Gotze et al., 2014a,b] disclose that SGX processors also employ a PUF, which generates a symmetric key that is used during the provisioning process. Specifically, at an early provisioning stage, the PUF key is encrypted with the GWK and transmitted to the key generation server. At a later stage, the key generation server encrypts the chip's fuse key material with the PUF key, and transmits it to the chip. The PUF key increases the cost of obtaining a chip's fuse key material, as an attacker must compromise both provisioning stages in order to be able to decrypt the fuse key material.

The SDM [Int, 2015b] mentions a 16-byte CPU security version number (SVN), which contains the version numbers of various TCB components, and is a source in the key derivation process. The patents further specify that the SVN register is made up of (most likely 8-bit) sections that contain the SVNs of each layer in the SGX initialization process, and that each initialization step sets the corresponding section to its SVN, and then locks it for the duration of the power-up cycle.

Intel's patents disclose that the key derivation process uses 128-bit AES in ECB mode as a pseudo-random function (PRF). When an SVN is an input to a key derivation process, a *PRF loop* is used, where the PRF is applied to a constant.

[Anati et al., 2013] confirms that the attestation uses Intel's EPID [Brickell and Li, 2009] group signature scheme.

The Intel patents indicate that EREPORT's KeyID is initialized to a random value on each processor power cycle, and is incremented after 2^{32} AES operations that use the value. They also indicate that each EREPORT may increment the KeyID by 1.

Neither the ISCA 2015 SGX tutorial nor the SDM mention the DAK described above. However, an Intel-sponsored book on the Management Engine (ME) [Ruan, 2014], which is the closest thing available to an official documentation for the ME, mentions that the ME contains e-fuses that store an EPID key which is used by the virtual TPM's software attestation feature.

Many aspects in the description of the ME's EPID key match the SGX patents' depiction of the DAK. Both sources use the same compressed representation of the EPID key. Also, both documents mention that the EPID key is encrypted with an AES key.

This coincidence, combined with the apparent lack of an SGX instruction that can be used to retrieve the DAK, prompts the consideration of the possibility that the DAK mentioned in the SGX patent is actually stored in the Intel ME's flash memory or e-fuses. This approach would make sense from a cost standpoint, as the ME's core uses a larger feature size than the CPU, so adding e-fuses or NVRAM to the ME die is cheaper than adding them to the CPU die.

While this approach seems attractive from a cost standpoint, if it were to be true, it would have significant implications on SGX's security properties. The DMI bus (§ I.2.9.1) that connects the CPU to the chipset that contains the ME is untrusted, so the EPID key cannot be transmitted between the CPU and the ME in plaintext. Below, we outline two possibilities for building a secure system where the DAK is stored in the ME's flash memory.

The most straightforward approach is having the Provision Enclave encrypt the DAK using the Provisioning Seal Key, as described in § I.5.8.2. Under this approach, the ME would effectively be an untrusted flash memory, so it would not be able to run the virtual TPM application described in [Ruan, 2014].

Another approach that preserves the virtual TPM functionality would entail having the ME authenticate the CPU as described in [Costan and Devadas, 2011], and using a key agreement protocol (§ I.3.2.2) to establish a secure communication channel over the DMI bus. In this case, the ME firmware is responsible for storing the DAK, as described in [Ruan, 2014], and can implement a virtual TPM.

The downside of the latter approach is that the ME must be counted as belonging to the SGX's TCB, as it is trusted to access the DAK. An attack that successfully plants malicious firmware into the ME, as described in § I.3.6 and briefly analyzed in § 2.7.5, would result in an exposed DAK, and a total compromise of SGX's security.

In its last chapter, [Ruan, 2014] mentions that the SGX architecture takes advantage of the ME using its Dynamic Application Loader (DAL) interface. This may refer to the Protected Audio/Video Path mentioned in an SGX paper [Hoekstra et al., 2013], or it may refer to the DAK provisioning methods speculated above.

2.7 SGX Security Properties

We have summarized SGX's programming model and the implementation details that are publicly documented in Intel's official documentation and published patents. We are now ready to bring this the information together in an analysis of SGX's security properties. We start the analysis by restating SGX's security guarantees, and spend the bulk of this section discussing how SGX fares when pitted against the attacks described in § I.3. We conclude the analysis with some troubling implications of SGX's lack of resistance to software side-channel attacks.

2.7.1 Overview

Intel's Software Guard Extensions (SGX) is Intel's latest iteration of a trusted hardware solution to the secure remote computation problem. The SGX design is centered around the ability to create an isolated container whose contents receives special hardware protections that are intended to translate into confidentiality, integrity, and freshness guarantees.

An enclave's initial contents is loaded by the system software on the computer, and therefore cannot contain secrets in plain text. Once initialized, an enclave is expected to participate in a software attestation process, where it authenticates itself to a remote server. Upon successful authentication, the remote server is expected to disclose some secrets to an enclave over a secure communication channel. The SGX design attempts to guarantee that the measurement presented during software attestation accurately represents the contents loaded into the enclave.

SGX also offers a certificate-based identity system that can be used to migrate secrets between enclaves that have certificates issued by the same authority. The migration process involves securing the secrets via authenticated encryption before handing them off to the untrusted system software, which passes them to another enclave that can decrypt them.

The same mechanism used for secret migration can also be used to cache the secrets obtained via software attestation in an untrusted storage medium managed by system software. This caching can reduce the number of times that the software attestation process needs to be performed in a distributed system. In fact, SGX's software attestation process is implemented by enclaves with special privileges that use the certificate-based identity system to securely store the CPU's attestation key in untrusted memory.

2.7.2 Physical Attacks

We begin by discussing SGX's resilience to the physical attacks described in § I.3.4. Unfortunately, this section is set to disappoint readers expecting definitive statements. The lack of publicly available details around the hardware implementation aspects of SGX precludes any rigorous analysis. However, we do know enough about SGX's implementation to point out a few avenues for future exploration.

Due to insufficient documentation, one can only hope that the SGX security model is not trivially circumvented by a port attack (§ I.3.4.1). We are particularly concerned about the Generic Debug eXternal Connection (GDXC) [Yuffe et al., 2011, Kurts et al., 2011], which collects

and filters the data transferred by the uncore's ring bus (§ I.2.11.3), and reports it to an external debugger.

The SGX memory protection measures are implemented at the core level, in the Page Miss Handler (PMH, § I.2.11.5) (§ 2.2) and at the chip die level, in the memory controller (§ 2.1.2). Therefore, the code and data inside enclaves is stored in plaintext in on-chip caches (§ I.2.11), which entails that the enclave contents travels without any cryptographic protection on the uncore's ring bus (§ I.2.11.3).

Fortunately, a recent Intel patent [Shanbhogue et al., 2015] indicates that Intel engineers are tackling at least some classes of attacks targeting debugging ports.

The SDM and SGX papers discuss the most obvious class of bus tapping attacks (§ I.3.4.2), which is the DRAM bus tapping attack. SGX's threat model considers DRAM and the bus connecting it to the CPU chip to be untrusted. Therefore, SGX's Memory Encryption Engine (MEE, § 2.1.2) provides confidentiality, integrity and freshness guarantees to the Enclave Page Cache (EPC, § I.5.1.1) data while it is stored in DRAM.

However, both the SGX papers and the ISCA 2015 tutorial on SGX admit that the MEE does not protect the addresses of the DRAM locations accessed when cache lines holding EPC data are evicted or loaded. This provides an opportunity for a malicious computer owner to observe an enclave's memory access patterns by combining a DRAM address line bus tap with carefully crafted system software that creates artificial pressure on the last-level cache (LLC ,§ I.2.11) lines that hold the enclave's EPC pages.

On a brighter note, as mentioned in § I.3.4.2, we are not aware of any successful DRAM address line bus tapping attack. Furthermore, SGX is vulnerable to cache timing attacks that can be carried out completely in software, so malicious computer owners do not need to bother setting up a physical attack to obtain an enclave's memory access patterns.

While the SGX documentation addresses DRAM bus tapping attacks, it makes no mention of the System Management bus (SMBus, § I.2.9.2) that connects the Intel Management En-

gine (ME, § I.2.9.2) to various components on the computer's motherboard.

In § 2.7.5, we will explain that the ME needs to be taken into account when evaluating SGX's memory protection guarantees. This makes us concerned about the possibility of an attack that taps the SMBus to reach into the Intel ME. The SMBus is much more accessible than the DRAM bus, as it has fewer wires that operate at a significantly lower speed. Unfortunately, without more information about the role that the Intel ME plays in a computer, we cannot move beyond speculation on this topic.

The threat model stated by the SGX design excludes physical attacks targeting the CPU chip (§ I.3.4.3). Fortunately, Intel's patents disclose an array of countermeasures aimed at increasing the cost of chip attacks.

For example, the original SGX patents [McKeen et al., 2009, Johnson et al., 2010] disclose that the Fused Seal Key and the Provisioning Key, which are stored in e-fuses (§ I.5.8.2), are encrypted with a *global wrapping logic key* (GWK). The GWK is a 128-bit AES key that is hard-coded in the processor's circuitry, and serves to increase the cost of extracting the keys from an SGX-enabled processor.

As explained in § I.3.4.3, e-fuses have a large feature size, which makes them relatively easy to "read" using a high-resolution microscope. In comparison, the circuitry on the latest Intel processors has a significantly smaller feature size, and is more difficult to reverse engineer. Unfortunately, the GWK is shared among all dies created from the same mask, so it suffers from the drawbacks of global secrets explained in § I.3.4.3.

Newer Intel patents [Gotze et al., 2014a,b] describe SGX-enabled processors that employ a *Physical Unclonable Function* (PUF), e.g., [Suh and Devadas, 2007], [Maes et al., 2009], which generates a symmetric key that is used during the provisioning process.

Specifically, at an early provisioning stage, the PUF key is encrypted with the GWK and transmitted to the key generation server. At a later stage, the key generation server encrypts the key material that will be burned into the processor chip's e-fuses with the PUF

key, and transmits the encrypted material to the chip. The PUF key increases the cost of obtaining a chip's fuse key material, as an attacker must compromise both provisioning stages in order to be able to decrypt the fuse key material.

As mentioned in previous sections, patents reveal design possibilities considered by the SGX engineers. However, due to the length of timelines involved in patent applications, patents necessarily describe earlier versions of the SGX implementation plans, which may not match the shipping implementation. We expect this may be the case with the PUF provisioning patents, as it makes little sense to include a PUF in a chip die and rely on e-fuses and a GWK to store SGX's root keys. Deriving the root keys from the PUF would be more resilient to chip imaging attacks.

SGX's threat model excludes power analysis attacks (§ I.3.4.4) and other side-channel attacks. This is understandable, as power attacks cannot be addressed at the architectural level. Defending against power attacks requires expensive countermeasures at the lowest levels of hardware implementation, which can only be designed by engineers who have deep expertise in both system security and Intel's manufacturing process. It follows that defending against power analysis attacks has a very high cost-to-benefit ratio.

2.7.3 Privileged Software Attacks

The SGX threat model considers system software to be untrusted. This is a prerequisite for SGX to qualify as a solution to the secure remote computation problem encountered by software developers who wish to take advantage of Infrastructure-as-a-Service (IaaS) cloud computing.

SGX's approach is also an acknowledgment of the realities of today's software landscape, where the system software that runs at high privilege levels (§ I.2.3) is so complex that security researchers constantly find vulnerabilities in it (§ I.3.5).

The SGX design prevents malicious software from directly reading or from modifying the EPC pages that store an enclave's code and data. This security property relies on two pillars in the SGX design.

First, the SGX implementation (§ 2.1) runs in the processor's microcode (§ I.2.14), which is effectively a higher privilege level that system software does not have access to. Along the same lines, SGX's security checks (§ 2.2) are the last step performed by the PMH, so they cannot be bypassed by any other architectural feature.

This implementation detail is only briefly mentioned in SGX's official documentation, but has a large impact on security. For context, Intel's Trusted Execution Technology (TXT, [Grawrock, 2009]), which is the predecessor of SGX, relied on Intel's Virtual Machine Extensions (VMX) for isolation. The approach was unsound, because software running in System Management Mode (SMM, § I.2.3) could bypass the restrictions used by VMX to provide isolation.

The security properties of SGX's memory protection mechanisms are discussed in detail in § 2.7.4.

Second, SGX's microcode is always involved when a CPU transitions between enclave code and non-enclave code (§ I.5.4), and therefore regulates all interactions between system software and an enclave's environment.

On enclave entry (§ I.5.4.1), the SGX implementation sets up the registers (§ I.2.2) that make up the execution state (§ I.2.6) of the logical processor (LP § I.2.9.4), so a malicious OS or hypervisor cannot induce faults in the enclave's software by tampering with its execution environment.

When an LP transitions away from an enclave's code due to a hardware exception (§ I.2.8.2), the SGX implementation stashes the LP's execution state into a State Save Area (SSA, § I.5.2.5) area inside the enclave and scrubs it, so the system software's exception handler cannot access any enclave secrets that may be stored in the execution state.

The protections described above apply to all levels of privileged software. SGX's transitions between an enclave's code and non-enclave code place SMM software on the same footing as the system software at lower privilege levels. System Management Interrupts (SMI, § I.2.12, § I.3.5), which cause the processor to execute SMM code, are handled using the same Asynchronous Enclave Exit (AEX, § I.5.4.3) process as all other hardware exceptions.

Reasoning about the security properties of SGX's transitions between enclave mode and non-enclave mode is very difficult. A correctness proof would have to take into account all features of the CPU that expose registers. Difficulty aside, such a proof would be very short-lived, because every generation of Intel CPUs tends to introduce new architectural features. The paragraph below gives a taste of what such a proof would look like.

`EENTER` (§ I.5.4.1) stores the RSP and RBP register values in the SSA used to enter the enclave, but stores XCR0 (§ I.2.6), FS and GS (§ I.2.7) in the non-architectural area of the TCS (§ 2.1.3). At first glance, it may seem elegant to remove this inconsistency and have `EENTER` store the contents of the XCR0, FS, and GS registers in the current SSA, along with RSP and RBP. However, this approach would break the Intel architecture's guarantees that only system software can modify XCR0, and application software can only load segment registers using selectors that index into the GDT or LDT set up by system software. Specifically, a malicious application could modify these privileged registers by creating an enclave that writes the desired values to the current SSA locations backing up the registers, and then executes `EEXIT` (§ I.5.4.2).

Unfortunately, the following sections will reveal that while SGX offers rather thorough guarantees against straightforward attacks on enclaves, its guarantees are almost non-existent when it comes to more sophisticated attacks, such as side-channel attacks. This section concludes by describing what may be the most egregious side-channel vulnerability in SGX.

Most modern Intel processors feature hyper-threading. On these CPUs, the execution units (§ I.2.10) and caches (§ I.2.11) on a core (§ I.2.9.4) are shared by two LPs, each of which has its own execution state. SGX does not prevent hyper-threading, so malicious system software can schedule a thread executing the code of a victim enclave on an LP that shares the core with an LP executing a snooping thread. This snooping thread can use the processor's high-resolution performance counter [Petters and Farber, 1999], in conjunction with microarchitectural knowledge of the CPU's execution units and out-

of-order scheduler, to learn the instructions executed by the victim enclave, as well as its memory access patterns.

This vulnerability can be fixed using two approaches. The straightforward solution is to require cloud computing providers to disable hyper-threading when offering SGX. The SGX enclave measurement would have to be extended to include the computer's hyper-threading configuration, so the remote parties in the software attestation process can be assured that their enclaves are hosted by a secure environment.

A more complex approach to fixing the hyper-threading vulnerability would entail having the SGX implementation guarantee that when an LP is executing an enclave's code, the other LP sharing its core is either inactive, or is executing the same enclave's code. While this approach is possible, its design would likely be quite cumbersome.

2.7.4 Memory Mapping Attacks

§ I.5.4 explained that the code running inside an enclave uses the same address translation process (§ I.2.5) and page tables as its host application. While this design approach makes it easy to retrofit SGX support into existing codebases, it also enables the address translation attacks described in § I.3.7.

The SGX design protects the code inside enclaves against the active attacks described in § I.3.7. These protections have been extensively discussed in prior sections, so we limit ourselves to pointing out SGX's answer to each active attack. We also explain the lack of protections against passive attacks, which can be used to learn an enclave's memory access pattern at 4KB page granularity.

SGX uses the Enclave Page Cache Map (EPCM, § I.5.1.2) to store each EPC page's position in its enclave's virtual address space. The EPCM is consulted by SGX's extensions to the Page Miss Handler (PMH, § 2.2.1), which prevent straightforward active address translation attacks (§ I.3.7.2) by rejecting undesirable address translations before they reach the TLB (§ I.2.11.5).

SGX allows system software to evict (§ I.5.5) EPC pages into untrusted DRAM, so that the EPC can be over-subscribed. The contents of the evicted pages and the associated EPCM metadata are pro-

tected by cryptographic primitives that offer confidentiality, integrity and freshness guarantees. This protects against the active attacks using page swapping described in § I.3.7.3.

When system software wishes to evict EPC pages, it must follow the process described in § I.5.5.1, which guarantees to the SGX implementation that all LPs have invalidated any TLB entry associated with pages that will be evicted. This defeats the active attacks based on stale TLB entries described in § I.3.7.4.

§ 2.3 outlines a correctness proof for the memory protection measures described above.

Unfortunately, SGX does not protect against passive address translation attacks (§ I.3.7.1), which can be used to learn an enclave's memory access pattern at page granularity. While this appears benign, recent work [Xu et al., 2015] demonstrates the use of these passive attacks in a few practical settings, which are immediately concerning for image processing applications.

The rest of this section describes the theory behind planning a passive attack against an SGX enclave. The reader is directed to [Xu et al., 2015] for a fully working system.

Passive address translation attacks rely on the fact that memory accesses issued by SGX enclaves go through the Intel architecture's address translation process (§ I.2.5), including delivering page faults (§ I.2.8.2) and setting the accessed (A) and dirty (D) attributes (§ I.2.5.3) on page table entries.

A malicious OS kernel or hypervisor can obtain the page-level trace of an application executing inside an enclave by setting the present (P) attribute to 0 on all of the enclave's pages before starting enclave execution. While an enclave executes, the malicious system software maintains exactly one instruction page and one data page present in the enclave's address space.

When a page fault is generated, CR2 contains the virtual address of a page accessed by enclave, and the error code indicates whether the memory access was a read or a write (bit 1) and whether the memory access is a data access or an instruction fetch access (bit 4). On a data access, the kernel tracing the enclave code's memory access

pattern would set the P flag of the desired page to 1, and set the P flag of the previously accessed data page to 0. Instruction accesses can be handled in a similar manner.

For a slightly more detailed trace, the kernel can set a desired page's writable (W) attribute to 0 if the page fault's error code indicates a read access, and only set it to 1 for write accesses. Also, applications that use a page as both code and data (self-modifying code and just-in-time compiling VMs) can be handled by setting a page's disable execution (XD) flag to 0 for a data access, and by carefully accounting for the case where the last accessed data page is the same as the last accessed code page.

Leaving an enclave via an Asynchronous Enclave Exit (AEX, § I.5.4.3) and re-entering the enclave via `ERESUME` (§ I.5.4.4) causes the CPU to flush TLB entries that contain enclave addresses, so a tracing kernel would not need to worry about flushing the TLB. The tracing kernel does not need to flush the caches either, because the CPU needs to perform address translation even for cached data.

A straightforward way to reduce this attack's power is to increase the page size, so the trace contains less information. However, the attack cannot be completely prevented without removing the kernel's ability to oversubscribe the EPC, which is a major benefit of paging.

2.7.5 Software Attacks on Peripherals

Since the SGX design does not trust the system software, it must be prepared to withstand the attacks described in § I.3.6, which can be carried out by the system software thanks to its ability to control peripheral devices on the computer's motherboard (§ I.2.9.1). This section summarizes the security properties of SGX when faced with these attacks, based on publicly available information.

When SGX is enabled on an LP, it configures the memory controller (MC, § I.2.11.3) integrated on the CPU chip die to reject any DMA transfer that falls within the Processor Reserved Memory (PRM, § I.5.1) range. The PRM includes the EPC, so the enclaves' contents is protected from the PCI Express attacks described in § I.3.6.1. This protection guarantee relies on the fact that the MC

is integrated on the processor's chip die, so the MC configuration commands issued by SGX's microcode implementation (§ 2.1.3) are transmitted over a communication path that never leaves the CPU die, and therefore can be trusted.

SGX regards DRAM as an untrusted storage medium, and uses cryptographic primitives implemented in the MEE to guarantee the confidentiality, integrity and freshness of the EPC contents that is stored into DRAM. This protects against software attacks on DRAM's integrity, like the rowhammer attack described in § I.3.6.2.

The SDM describes an array of measures that SGX takes to disable processor features intended for debugging when a LP starts executing an enclave's code. For example, enclave entry (§ I.5.4.1) disables Precise Event Based Sampling (PEBS) for the LP, as well as any hardware breakpoints placed inside the enclave's virtual address range (ELRANGE, § I.5.2.1). This addresses some of the attacks described in § I.3.6.3, which take advantage of performance monitoring features to get information that typically requires access to hardware probes.

At the same time, the SDM does not mention anything about uncore PEBS counters, which can be used to learn about an enclave's LLC activity. Furthermore, the ISCA 2015 tutorial slides mention that **SGX does not protect against software side-channel attacks** that rely on performance counters.

This limitation in SGX's threat model leaves security-conscious enclave authors in a rather terrible situation. These authors know that SGX does not protect their enclaves against a class of software attacks. At the same time, they cannot even contemplate attempting to defeat these attacks on their own, due to lack of information. Specifically, the documentation that is publicly available from Intel does not provide enough information to model the information leakage due to performance counters.

For example, Intel does not document the mapping implemented in CBoxes (§ I.2.11.3) between physical DRAM addresses and the LLC slices used to cache the addresses. This mapping impacts several uncore performance counters, and the impact is strong enough to allow security researches to reverse-engineer the mapping [Inci et al., 2015,

Maurice et al., 2015, Yarom et al., 2015]. Therefore, it is safe to assume that a malicious computer owner who knows the CBox mapping can use the uncore performance counters to learn about an enclave's memory access patterns.

The SGX papers mention that SGX's threat model includes attacks that overwrite the flash memory chip that stores the computer's firmware, which result in malicious code running in SMM. However, the official SGX documentation is silent about the implications of an attack that compromises the firmware executed by the Intel ME.

§ I.3.6.4 states that the ME's firmware is stored in the same flash memory as the boot firmware, and enumerates some of ME's special privileges that enable it to help system administrators remotely diagnose and fix hardware and software issues. Given that the SGX design is concerned about the possibility of malicious computer firmware, it is reasonable to be concerned about malicious ME firmware.

§ I.3.6.4 argues that an attacker who compromises the ME can carry out actions that are usually classified as physical attacks. An optimistic security researcher can observe that the most scary attack vector afforded by an ME takeover appears to be direct DRAM access, and SGX already assumes that the DRAM is untrusted. Therefore, an ME compromise would be equivalent to the DRAM attacks analyzed in § 2.7.2.

However, we are troubled by the lack of documentation on the ME's implementation, as certain details are critical to SGX's security analysis. For example, the ME is involved in the computer's boot process (§ I.2.13, § I.2.14.4), so it is unclear if it plays any part in the SGX initialization sequence. Furthermore, during the security boot stage (SEC, § I.2.13.2), the bootstrap LP (BSP) is placed in Cache-As-Ram (CAR) mode so that the PEI firmware can be stored securely while it is measured. This suggests that it would be convenient for the ME to receive direct access to the CPU's caches, so that the ME's TPM implementation can measure the firmware directly. At the same time, a special access path from the ME to the CPU's caches may sidestep the MEE, allowing an attacker who has achieved ME code execution to directly read the EPC's contents.

2.7.6 Cache Timing Attacks

The SGX threat model excludes the cache timing attacks described in § I.3.8. The SGX documentation bundles these attacks together with other side-channel attacks and summarily dismisses them as complex physical attacks. However, cache timing attacks can be mounted entirely by unprivileged software running at ring 3. This section describes the implications of SGX's environment and threat model on cache timing attacks.

The main difference between SGX and a standard architecture is that SGX's threat model considers the system software to be untrusted. As explained earlier, this accurately captures the situation in remote computation scenarios, such as cloud computing. SGX's threat model implies that the system software can be carrying out a cache timing attack on the software inside an enclave.

A malicious system software translates into significantly more powerful cache timing attacks, compared to those described in § I.3.8. The system software is in charge of scheduling threads on LPs, and also in charge of setting up the page tables used by address translation (§ I.2.5), which control cache placement (§ I.2.11.5).

For example, the malicious kernel set out to trace an enclave's memory access patterns described in § 2.7.4 can improve the accuracy of a cache timing attack by using page coloring [Kessler and Hill, 1992] principles to partition [Lin et al., 2008] the cache targeted by the attack. In a nutshell, the kernel divides the cache's sets (§ I.2.11.2) into two regions, as shown in Figure 2.9.

The system software stores the whole of the victim enclave's code and data in DRAM addresses that map to the cache sets in one of the regions, and stores its own code and data in DRAM addresses that map to the other region's cache sets. The snooping thread's code is assumed to be a part of the OS. For example, in a typical 256 KB (per-core) L2 cache organized as 512 8-way sets of 64-byte lines, the tracing kernel could allocate lines 0-63 for the enclave's code page, lines 64-127 for the enclave's data page, and use lines 128-511 for its own pages.

To the best of our knowledge, there is no minor modification to SGX that would provably defend against cache timing attacks. How-

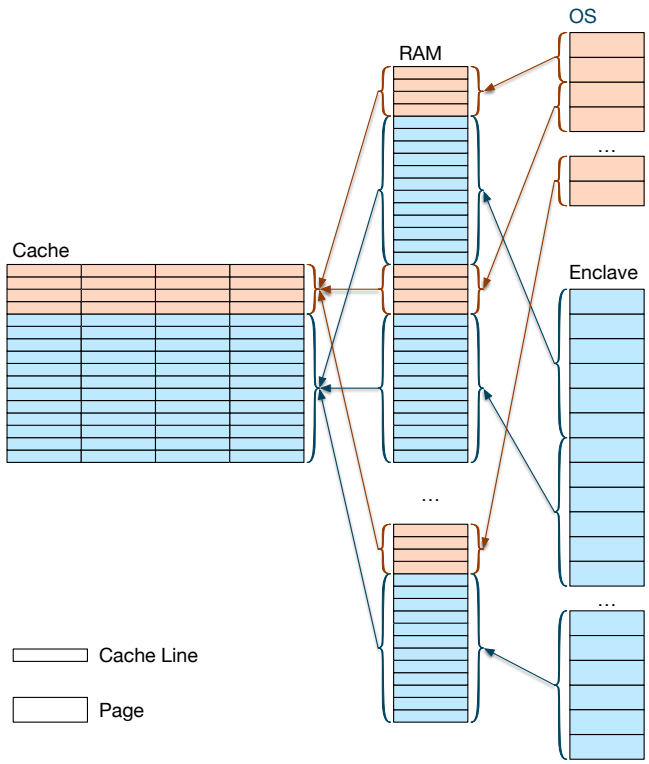

Figure 2.9: A malicious OS can partition a cache between the software running inside an enclave and its own malicious code. Both the OS and the enclave software have cache sets dedicated to them. When allocating DRAM to itself and to the enclave software, the malicious OS is careful to only use DRAM regions that map to the appropriate cache sets. On a system with an Intel CPU, the the OS can partition the L2 cache by manipulating the page tables in a way that is completely oblivious to the enclave's software.

ever, the SGX design could take a few steps to increase the cost of cache timing attacks. For example, SGX's enclave entry implementation could flush the core's private caches, which would prevent cache timing attacks from targeting them. This measure would defeat the cache timing attacks described below, and would only be vulnerable to more sophisticated attacks that target the shared LLC, such as [Yarom and Falkner, 2013, Liu et al., 2015]. The description above assumes that multi-threading has been disabled, for the reasons explained in § 2.7.3.

Barring the additional protection measures described above, a tracing kernel can extend the attack described in § 2.7.4 with the steps outlined below to take advantage of cache timing and narrow down the addresses in an application's memory access trace to cache line granularity.

Right before entering an enclave via `EENTER` or `ERESUME`, the kernel would issue `CLFLUSH` instructions to flush the enclave's code page and data page from the cache. The enclave could have accessed a single code page and a single data page, so flushing the cache should be reasonably efficient. The tracing kernel then uses 16 bogus pages (8 for the enclave's code page, and 8 for the enclave's data page) to load all 8 ways in the 128 cache sets allocated by enclave pages. After an AEX gives control back to the tracing kernel, it can read the 16 bogus pages, and exploit the time difference between an L2 cache hit and a miss to see which cache lines were evicted and replaced by the enclave's memory accesses.

An extreme approach that can provably defeat cache timing attacks is disabling caching for the PRM range, which contains the EPC. The SDM is almost completely silent about the PRM, but the SGX manuals that it is based on state that the allowable caching behaviors (§ I.2.11.4) for the PRM range are uncacheable (UC) and write-back (WB). This could become useful if the SGX implementation would make sure that the PRM's caching behavior cannot be changed while SGX is enabled, and if the selected behavior would be captured by the enclave's measurement (§ I.5.6).

2.7.7 Software Attacks on Private Microarchitectural State

In addition to straightforward leaks of private information via page table metadata, and well-studied cache timing attacks, SGX also fails to protect enclaves against an adversary seeking to infer control flow information via microarchitectural state maintained by hardware. Software scheduled onto a where an enclave was (or is) executing instructions shares persistent structures with private execution.

While the reorder buffer and other transient state can be isolated by disabling hyper-threading, and the TLB is protected by SGX, the enclave's branch history is vulnerable [Lee et al., 2016]. Preventing this attack would require the SGX microcode to flush branch history when entering and exiting enclave mode (and preventing untrusted threads from sharing a physical core with an enclave), which it does not do at the time of this writing. The state of the DRAM controller (selected bank, etc.) likewise leaks enclaves' confidential DRAM accesses [Wang et al., 2017].

2.7.8 Other Software Side-Channel Attacks and SGX

The SGX design reuses a few terms from the Trusted Platform Module (TPM, § I.4.4) design. This helps software developers familiar with TPM understand SGX faster. At the same time, the term reuse invites the assumption that SGX's software attestation is implemented in tamper-resistant hardware, similarly to the TPM design.

§ I.5.8 explains that, in fact, the SGX design delegates the creation of attestation signatures to software that runs inside a Quoting Enclave with special privileges that allows it to access the processor's attestation key. Restated, SGX includes an enclave whose software reads the attestation key and produces attestation signatures.

Creating the Quoting Enclave is a very elegant way of reducing the complexity of the hardware implementation of SGX, assuming that the isolation guarantees provided by SGX are sufficient to protect the attestation key. However, the security analysis in § 2.7 reveals that enclaves are vulnerable to a vast array of software side-channel at-

tacks, which have been demonstrated effective in extracting a variety of secrets from isolated environments.

The gaps in the security guarantees provided to enclaves place a large amount of pressure on Intel's software developers, as they must attempt to implement the EPID signing scheme used by software attestation without leaking any information. Intel's ISCA 2015 SGX tutorial slides suggest that the SGX designers will advise developers to write their code in a way that avoids data-dependent memory accesses, as suggested in § I.3.8.4, and perhaps provide analysis tools that detect code that performs data-dependent memory accesses.

The main drawback of the approach described above is that it is extremely cumbersome. § I.3.8.4 describes that, while it may be possible to write simple pieces of software in such a way that they do not require data-dependent memory accesses, there is no known process that can scale this to large software systems. For example, each virtual method call in an object-oriented language results in data-dependent code fetches.

The ISCA 2015 SGX tutorial slides also suggest that the efforts of removing data-dependent memory accesses should focus on cryptographic algorithm implementations, in order to protect the keys that they handle. This is a terribly misguided suggestion, because cryptographic key material has no intrinsic value. Attackers derive benefits from obtaining the data that is protected by the keys, such as medical and financial records.

Some security researchers focus on protecting cryptographic keys because they are the target of today's attacks. Unfortunately, it is easy to lose track of the fact that keys are being attacked simply because they are the lowest hanging fruit. A system that can only protect the keys will have a very small positive impact, as the attackers will simply shift their focus on the algorithms that process the valuable information, and use the same software side-channel attacks to obtain that information directly.

The second drawback of the approach described towards the beginning of this section is that while eliminating data-dependent memory accesses should thwart the attacks described in § 2.7.4 and § 2.7.6,

the measure may not be sufficient to prevent the hyper-threading attacks described in § 2.7.3. The level of sharing between the two logical processors (LP, § I.2.9.4) on the same CPU core is so high that it is possible that a snooping LP can learn more than the memory access pattern from the other LP on the same core.

For example, if the number of cycles taken by an integer ALU to execute a multiplication or division micro-op (§ I.2.10) depends on its inputs, the snooping LP could learn some information about the numbers multiplied or divided by the other LP. While this may be a simple example, it is safe to assume that the Quoting Enclave will be studied by many motivated attackers, and that any information leak will be exploited.

3

The MIT Sanctum Processor

Sanctum offers the same promise as Intel's Software Guard Extensions (SGX), namely strong provable isolation of software modules running concurrently and sharing resources, but protects against an important class of additional software attacks that infer private information from a program's memory access patterns. Sanctum shuns unnecessary complexity, leading to a simpler security analysis. The Sanctum processor design follows a principled approach to eliminating entire attack surfaces through isolation, rather than repairing attack-specific privacy leaks. Most of Sanctum's logic is implemented in trusted software, which does not perform cryptographic operations using keys, and is easier to analyze than SGX's opaque microcode, which does. The Sanctum prototype targets a Rocket RISC-V core [Lee et al., 2014], an open implementation that allows any researcher to reason about its security properties. Sanctum's extensions can be adapted to other processors, as the design does not modify any major CPU building block. Instead, Sanctum adds hardware at interfaces between generic building blocks, enforcing invariants to uphold Sanctum's security policy without impacting cycle time. Sanctum demonstrates that strong software isola-

tion is achievable with a surprisingly small set of minimally invasive hardware changes, and a very reasonable design and performance cost.

3.1 Threat Model

Sanctum isolates the software inside an **enclave** from other software on the same computer. All outside software, including privileged system software, can only interact with an enclave via a small set of primitives provided by the security monitor. Programmers are expected to move the sensitive code in their applications into enclaves. In general, an enclave receives encrypted sensitive information from outside, decrypts the information and performs some computation on it, and then returns encrypted results to the outside world.

For example, medical imaging software would use an enclave to decrypt a patient's X-ray and produce a diagnostic via an image processing algorithm. Application code that does not handle sensitive data, such as receiving encrypted X-rays over the network and storing the encrypted images in a database, would not be enclaved.

Sanctum assumes that an attacker can compromise any operating system and hypervisor present on the computer executing the enclave, and can launch rogue enclaves. The attacker knows the target computer's architecture and micro-architecture. The attacker can analyze passively collected data, such as page fault addresses, as well as mount active attacks, such as direct or DMA memory probing, and cache timing attacks.

Sanctum's isolation protects the integrity and privacy of the code and data inside an enclave against any practical **software** attack that relies on observing or interacting with the enclave software via means outside the interface provided by the security monitor. In other words, Sanctum does not protect enclaves that leak their own secrets directly (e.g., by writing to untrusted memory) or by timing their operations (e.g., by modulating their completion times). In effect, Sanctum solves the security problems that emerge from sharing a computer among mutually distrusting applications.

This distinction is particularly subtle in the context of cache timing attacks. Sanctum does not protect against attacks like the one in [Brumley and Boneh, 2005], where the victim application leaks information via its public API, and the leak occurs even if the victim runs on a dedicated machine. Sanctum *does* protect against attacks like Flush+Reload [Yarom and Falkner, 2013], which exploit shared hardware resources to interact with the victim via methods outside its public API.

Sanctum also defeats attackers who aim to compromise an OS or hypervisor by running malicious applications and enclaves. This addresses concerns that enclaves provide new attack vectors for malware [Rutkowska, 2013, Davenport, 2014]. Sanctum assumes that the benefits of meaningful software isolation outweigh enabling a new avenue for frustrating malware detection and reverse engineering [Dunn et al., 2011].

Lastly, Sanctum protects against a malicious computer owner who attempts to lie about the security monitor running on the computer. Specifically, the attacker aims to obtain an attestation stating that the computer is running an uncompromised security monitor, whereas a different monitor had been loaded in the boot process. The uncompromised security monitor must not have any known vulnerability that causes it to disclose its cryptographic keys. The attacker is assumed to know the target computer's architecture and micro-architecture, and is allowed to run any combination of malicious security monitor, hypervisor, OS, applications and enclaves.

Sanctum does not prevent timing attacks that exploit bottlenecks in the cache coherence directory bandwidth or in the DRAM bandwidth, deferring these to future work. The *latency* of memory operations exhibited by high-end DRAM controllers, however, has been shown a viable threat surface for timing attacks [Pessl et al., 2015], including in the context of SGX enclaves [Wang et al., 2017], meaning Sanctum enclaves must map DRAM regions to disjoint DRAM banks (this is the responsibility of the OS, and is checked during attestation), or implement a constant access latency DRAM controller, in order to guarantee confidentiality of DRAM accesses. In general, any component of the system

that exhibits variable timing due to the behavior of software must be considered a potential threat surface and addressed appropriately.

Sanctum does not protect against denial-of-service (DoS) attacks by compromised system software: a malicious OS may deny service by refusing to allocate any resources to an enclave. Sanctum *does* protect against malicious enclaves attempting to DoS an uncompromised OS.

Sanctum assumes correct underlying hardware, so it does not protect against software attacks that exploit hardware bugs (fault-injection attacks), such as rowhammer [Kim et al., 2014, Seaborn and Dullien, 2015].

Sanctum's isolation mechanisms exclusively target software attacks. § 3.8 mentions related work that can harden a Sanctum system against some physical attacks. Furthermore, software attacks that rely on sensor data are considered to be physical attacks. For example, Sanctum does not address information leakage due to power variations, because software would require a temperature or current sensor to carry out such an attack.

3.2 Programming Model Overview

By design, Sanctum's programming model minimally deviates from SGX, while providing stronger security guarantees. The expectation is that application authors will link against a Sanctum-aware runtime that abstracts away most aspects of Sanctum's programming model. For example, C programs would use a modified implementation of the `libc` standard library. This section assumes the reader's familiarity with SGX as described in I.5 and Section 2.

The software stack on a Sanctum machine, shown in Figure 3.1, resembles the SGX stack with one notable exception: SGX's microcode is replaced by a trusted software component, the **security monitor**, which is protected from compromised system software, as it runs at the highest privilege level (machine level in RISC-V).

Sanctum relegates the management of computation resources, such as DRAM and execution cores, to untrusted system software (as does SGX). In Sanctum, the security monitor checks the system software's

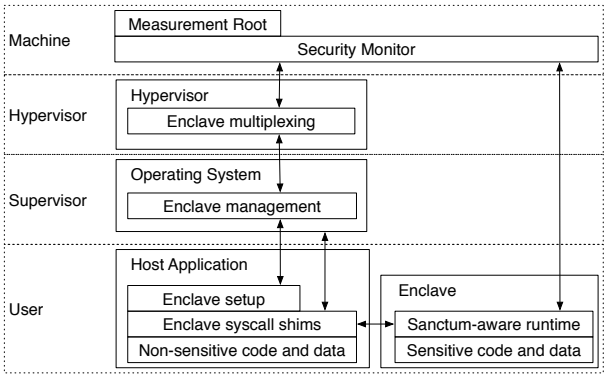

Figure 3.1: Software stack on a Sanctum machine; The blue text represents additions required by Sanctum. The bolded elements are in the software TCB.

allocation decisions for correctness and commits them into the hardware's configuration registers. For simplicity, we refer to the software that manages resources as an *OS (operating system)*, even though it may be a combination of a hypervisor and a guest OS kernel.

Figure 3.1 is representative of today's popular software stacks, where an operating system handles scheduling and demand paging, and the hypervisor multiplexes the computer's CPU cores. Sanctum is easy to integrate in such a stack, because the API calls that make up the security monitor's interface were designed with multiplexing in mind. Furthermore, a security-conscious hypervisor can use Sanctum's cache isolation primitive (DRAM region) to protect against cross-VM cache timing attacks [Apecechea et al., 2014].

An enclave stores its code and private data in parts of DRAM that have been allocated by the OS exclusively for the enclave's use (as does SGX), which are collectively called **the enclave's memory**. Consequently, we refer to the regions of DRAM that are not allocated to any enclave as **OS memory**. The security monitor tracks DRAM ownership, and ensures that no piece of DRAM is assigned to more than one enclave.

Each Sanctum enclave uses a range of virtual memory addresses (EVRANGE) to access its memory. The enclave's memory is mapped by the enclave's own page tables, which are stored in the enclave's mem-

ory (Figure 3.2). This makes private the page table dirty and accessed bits, which can reveal memory access patterns at page granularity. Exposing an enclave's page tables to the untrusted OS leaves the enclave vulnerable to attacks such as [Xu et al., 2015].

The enclave's virtual address space outside EVRANGE is used to access its host application's memory, via the page tables set up by the OS. Sanctum's hardware extensions implement dual page table lookup (§ 3.5.2), and make sure that an enclave's page tables can only point into the enclave's memory, while OS page tables can only point into OS memory (§ 3.5.3).

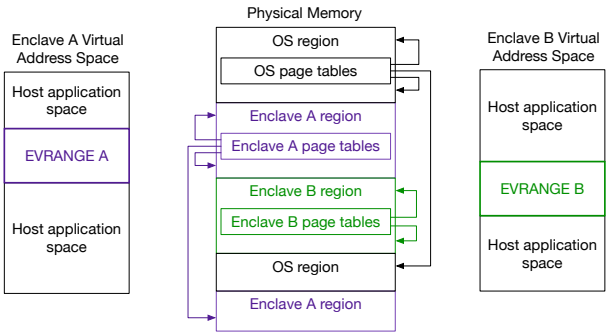

Figure 3.2: Per-enclave page tables.

Sanctum supports multi-threaded enclaves, and enclaves must appropriately provision for thread state data structures. Enclave threads, like their SGX cousins, run at the lowest privilege level (user level in RISC-V), meaning a malicious enclave cannot compromise the OS. Specifically, enclaves may not execute privileged instructions; address translations that use OS page tables generate page faults when accessing supervisor pages.

The per-enclave metadata used by the security monitor is stored in dedicated DRAM regions (**metadata regions**), each managed at the page level by the OS, and each includes a page map that is used by the security monitor to verify the OS' decisions (much like the EPC and EPCM in SGX, respectively). Unlike SGX's EPC, the metadata region pages only store enclave and thread metadata. Figure 3.3 shows how these structures are weaved together.

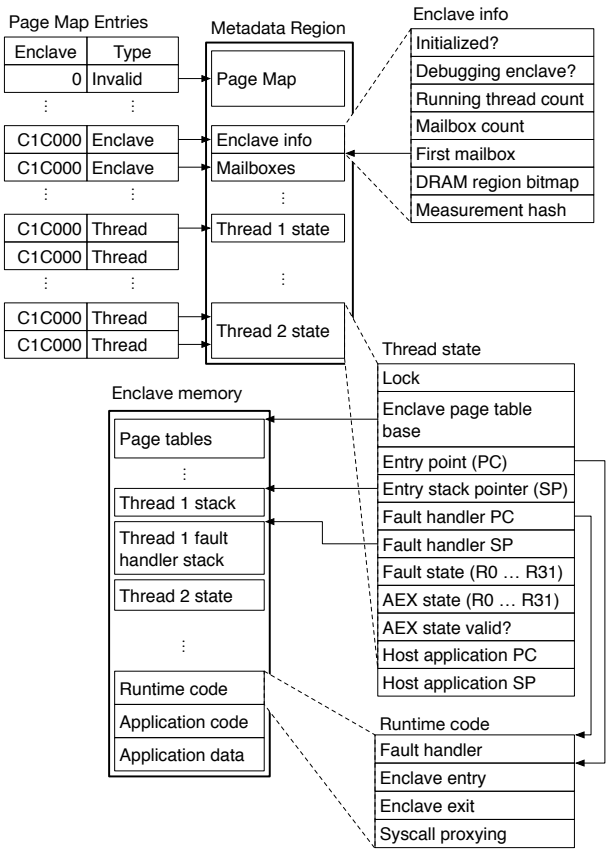

Figure 3.3: Enclave layout and data structures.

Sanctum considers system software to be untrusted, and governs transitions into and out of enclave code. An enclave's host application **enters an enclave** via a security monitor call that locks a thread state area, and transfers control to its entry point. After completing its intended task, the enclave code **exits** by asking the monitor to unlock the thread's state area, and transfer control back to the host application.

Enclaves cannot make system calls directly: Sanctum cannot trust the OS to restore an enclave's execution state, so the enclave's runtime must ask the host application to proxy syscalls such as file system and network I/O requests.

Sanctum's security monitor is the first responder for interrupts: an interrupt received during enclave execution causes an *asynchronous enclave exit* (AEX), whereby the monitor saves the core's registers in the current thread's AEX state area, zeroes the registers, sanitizes core microarchitectural state (via a fixed sequence of branches and loads), exits the enclave, and dispatches the interrupt as if it was received by the code entering the enclave.

On a normal enclave enter or exit operation (as opposed to an AEX), the enclave is expected to sanitize its own microarchitectural state, including branch history and private caches via a sequence of unprivileged instructions.

Unlike SGX, resuming enclave execution after an AEX means reentering the enclave using its normal entry point, and having the enclave's code ask the security monitor to restore the pre-AEX execution state. Sanctum enclaves are aware of asynchronous exits so they can implement security policies. For example, an enclave thread that performs time-sensitive work, such as periodic I/O, may terminate itself if it ever gets preempted by an AEX.

The security monitor configures the CPU to dispatch all faults occurring within an enclave directly to the enclave's designated fault handler, which is expected to be implemented by the enclave's runtime (SGX sanitizes and dispatches faults to the OS). For example, a `libc` runtime would translate faults into UNIX signals which, by default, would exit the enclave. It is possible, though not advisable for performance reasons (§ 3.6.3), for a runtime to handle page faults and implement demand paging securely, and robust against the attacks described in [Xu et al., 2015].

Unlike SGX, Sanctum isolates each enclave's data throughout the system's cache hierarchy. The security monitor flushes per-core caches, such as the L1 cache and the TLB, whenever a core jumps between enclave and non-enclave code. *Last-level cache* (LLC) isolation is achieved by a simple partitioning scheme supported by Sanctum's hardware extensions (§ 3.5.1).

Sanctum's isolation is also stronger than SGX's with respect to fault handling. While SGX sanitizes the information that an OS re-

ceives during a fault, Sanctum achieves full isolation by having the security monitor route the faults that occur inside an enclave to that enclave's fault handler. This removes all information leaks via the fault timing channel.

Sanctum's strong isolation yields a simple security model for application developers: *all computation that executes inside an enclave, and only accesses data inside the enclave, is protected from any attack mounted by software outside the enclave.* All communication with the outside world, including accesses to non-enclave memory, is subject to attacks.

Sanctum assumes that the enclave runtime implements the security measures needed to protect the enclave's communication with other software modules. For example, any algorithm's memory access patterns can be protected by ensuring that the algorithm only operates on enclave data. The runtime can implement this protection simply by copying any input buffer from non-enclave memory into the enclave before computing on it.

The enclave runtime can use Native Client's approach [Yee et al., 2009] to ensure that the rest of the enclave software only interacts with the host application via the runtime to mitigate potential security vulnerabilities in enclave software.

The lifecycle of a Sanctum enclave closely resembles the lifecycle of its SGX equivalent. An enclave is created when its host application performs a system call asking the OS to create an enclave from a dynamically loadable module (`.so` or `.dll` file). The OS invokes the security monitor to assign DRAM resources to the enclave, and to load the initial code and data pages into the enclave. Once the complete set of pages of an enclave are loaded, the enclave is marked as initialized via another security monitor call.

Sanctum's software attestation scheme is a simplified version of SGX's scheme, and reuses a subset of its concepts. The data used to initialize an enclave is cryptographically hashed, yielding the enclave's *measurement*. An enclave can invoke a secure inter-enclave messaging service to send a message to a privileged *attestation enclave*

that can access the security monitor's attestation key, and produces the attestation signature.

3.3 Protection Boundaries

Sanctum's isolation protects the privacy and integrity of an enclave's software, even in the face of a malicious operating system. Sanctum improves upon SGX by isolating cache sets and page tables used to access an enclave's private memory, as well as microarchitectural state updated as a side effect of enclave execution. The improved isolation defeats attacks that exploit the memory access pattern information leaks that result from cache and page table sharing, as well as attacks attempting to infer private control flow information from observing core state after enclave execution.

3.4 Security Primitives

Sanctum uses strong isolation to defeat information leaks. Enclaves that execute concurrently on different cores are isolated in the last-level cache (LLC) using a simple partitioning scheme (§ 3.5.1). Page table sharing is prevented via enclaves mapping their own physical memory via private page tables inaccessible to other software (§ 3.5.2). Core microarchitectural state including private cache contents, branch history, and TLBs is kept private by the security monitor, which interposes on each context switch involving an enclave, and sanitizes these structures.

Sanctum's hardware modifications target the DMA master (§ 3.5.4) and the interfaces to the last-level cache (LLC) (shown in Figure 3.8), as well as the interfaces between the memory management unit's (MMU) page walker, the translation lookaside buffers (TLBs), and the L1 data cache (shown in Figure 3.11).

Sanctum interposes on the interface between the LLC and the core-private caches to tweak the mapping between physical addresses and LLC sets, so that the computer's DRAM is split into many equal-sized regions, and the addresses in each DRAM region use distinct LLC sets (§ 3.5.1). Sanctum augments the interface between the TLBs and the MMU page walker with registers supporting per-enclave page

tables (§ 3.5.2), and Sanctum adds some logic to the interface between the page walker and the L1 cache to provide a method for constraining a page table to a set of DRAM regions (§ 3.5.3). Sanctum modifies the DMA master to reject DMA transfers that fall outside a safe range of memory addresses set by the security monitor (§ 3.5.4). Sanctum allows DRAM banks to be exclusively allocated to enclaves, or it must implements a DRAM controller with constant expected access latency in order to protect the confidentiality of enclaves' interactions with the DRAM controller.

Sanctum authenticates enclaves using the same principles as earlier secure processors such as Aegis [Suh et al., 2003] and SGX. Each Sanctum processor has an asymmetric key pair, and a certificate from the manufacturer for its public key. After an enclave is started, it can obtain an attestation intended to convince a remote party that it is communicating to that specific enclave running in a trusted environment. The attestation is a signature chain that starts at the manufacturer's trusted root key, and ends with a signature that covers the remote party's challenge nonce, the enclave's measurement (a cryptographic hash of the enclave's initial state), and a value produced by the enclave, which is generally used to start a key exchange protocol such as Diffie-Hellman [Diffie and Hellman, 1976].

Sanctum's attestation chain starts with the asymmetric key pair built into the processor. The next link in the chain is the *measurement root* (§ 3.6.1), a piece of trusted software that is burned into the processor's ROM. The measurement root contains the first instructions executed by a processor after it is powered on or reset, and its main job is to compute a measurement of the security monitor (a cryptographic hash) and add it to the attestation chain. The monitor produces enclave attestations.

3.5 Hardware Modifications

3.5.1 LLC Address Input Transformation

Figure 3.4 depicts a physical address in a toy computer with 32-bit virtual addresses and 21-bit physical addresses, 4,096-byte pages, a set-associative LLC with 512 sets and 64-byte lines, and 256 KB of DRAM.

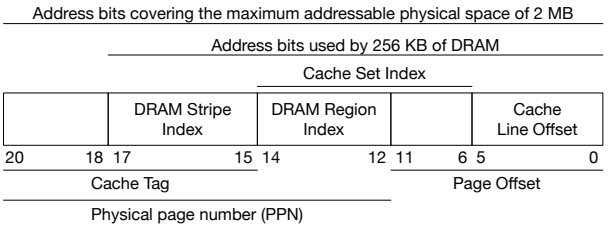

Figure 3.4: Interesting bit fields in a physical address.

The location where a byte of data is cached in the LLC depends on the low-order bits in the byte's physical address. The *set index* determines which of the LLC lines can cache the line containing the byte, and the *line offset* locates the byte in its cache line. A virtual address's low-order bits make up its *page offset*, while the other bits are its *virtual page number* (VPN). Address translation leaves the page offset unchanged, and translates the VPN into a *physical page number* (PPN), based on the mapping specified by the page tables.

Sanctum defines the **DRAM region index** in a physical address as the intersection between the PPN bits and the cache index bits. This is the maximal set of bits that impact cache placement *and* are determined by privileged software via page tables. Sanctum defines a **DRAM region** to be the subset of DRAM with addresses having the same DRAM region index. In Figure 3.4, for example, address bits $[14 \ldots 12]$ are the DRAM region index, dividing the physical address space into 8 DRAM regions.

In a typical system without Sanctum's hardware extensions, DRAM regions are made up of multiple continuous **DRAM stripes**, where each stripe is exactly one page long. The top of Figure 3.5 drives this point home, by showing the partitioning of our toy computer's 256 KB

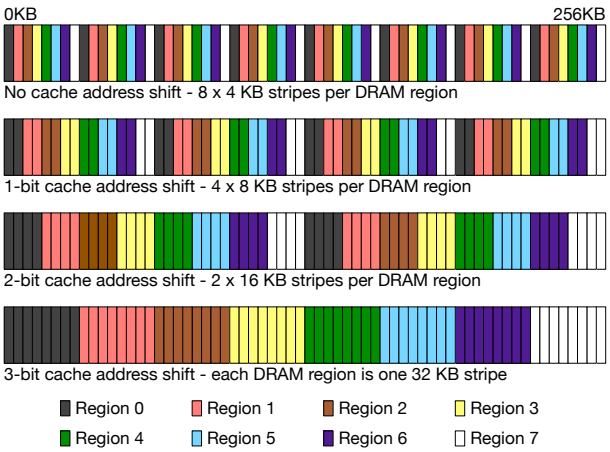

Figure 3.5: Address shift for contiguous DRAM regions.

of DRAM into DRAM regions. The fragmentation of DRAM regions makes it difficult for the OS to allocate contiguous DRAM buffers, which are essential to the efficient DMA transfers used by high performance devices. In our example, if the OS only owns 4 DRAM regions, the largest contiguous DRAM buffer it can allocate is 16 KB.

Observe that, up to a certain point, circularly shifting (rotating) the PPN of a physical address to the right by one bit, before it enters the LLC, doubles the size of each DRAM stripe and halves the number of stripes in a DRAM region, as illustrated in Figure 3.5.

Sanctum takes advantage of this effect by adding a **cache address shifter** that circularly shifts the PPN to the right by a certain amount of bits, as shown in Figures 3.6 and 3.8. In our example, configuring the cache address shifter to rotate the PPN by 3 yields contiguous DRAM regions, so an OS that owns 4 DRAM regions could hypothetically allocate a contiguous DRAM buffer covering half of the machine's DRAM.

The cache address shifter's configuration depends on the amount of DRAM present in the system. If our example computer could have 128 KB - 1 MB of DRAM, its cache address shifter must support shift amounts from 2 to 5. Such a shifter can be implemented via a 3-position variable shifter circuit (series of 8-input MUXes), and a fixed shift by 2 (no logic), as illustrated by Figure 3.7. Alternatively, in systems with

known DRAM configuration (embedded, SoC, etc.), the shift amount can be fixed, and implemented with no logic.

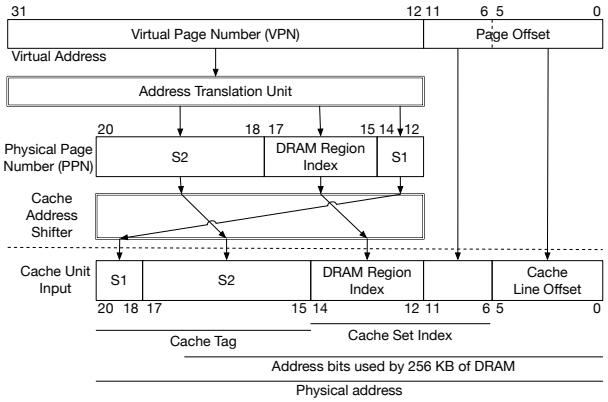

Figure 3.6: Cache address shifter, 3 bit PPN rotation.

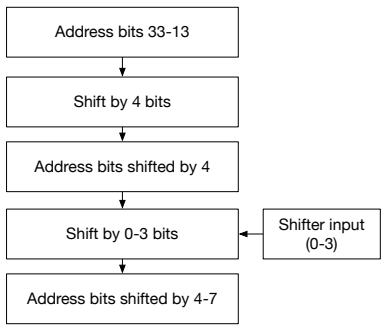

Figure 3.7: A variable shifter that can shift by 2-5 bits can be composed of a fixed shifter by 2 bits and a variable shifter that can shift by 0-3 bits.

3.5.2 Page Walker Input

Sanctum's per-enclave page tables require an enclave page table base register `eptbr` that stores the physical address of the currently running enclave's page tables, and has similar semantics to the page table base register `ptbr` pointing to the operating system-managed page tables. These registers may only be accessed by the Sanctum security monitor,

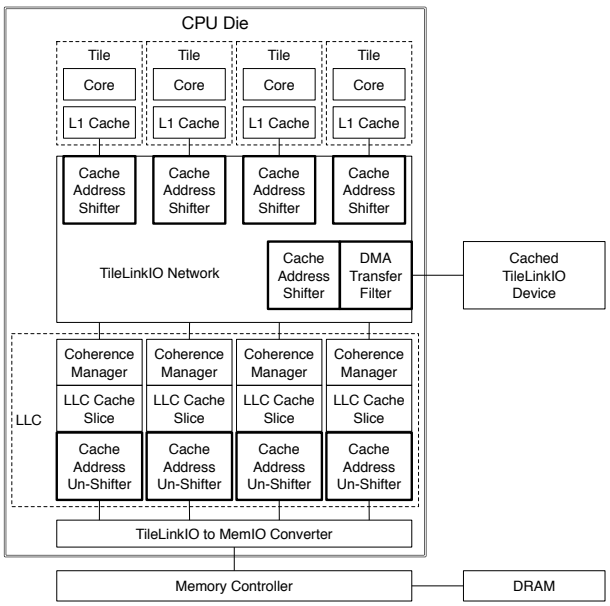

Figure 3.8: Sanctum's cache address shifter and DMA transfer filter logic in the context of a Rocket uncore.

which provides an API call for the OS to modify `ptbr`, and ensures that `eptbr` always points to the current enclave's page tables.

The circuitry handling TLB misses switches between `ptbr` and `eptbr` based on two registers that indicate the current enclave's EVRANGE, namely `evbase` (enclave virtual address space base) and `evmask` (enclave virtual address space mask). When a TLB miss occurs, the circuit in Figure 3.9 selects the appropriate page table base by ANDing the faulting virtual address with the mask register and comparing the output against the base register. Depending on the comparison result, either `eptbr` or `ptbr` is forwarded to the page walker as the page table base address.

In addition to the page table base registers, Sanctum uses 3 more pairs of registers that will be described in the next section. On a 64-bit RISC-V computer, the modified FSM input requires 5 extra 52-bit registers (the bottom 12 bits in a 64-bit page-aligned address will always

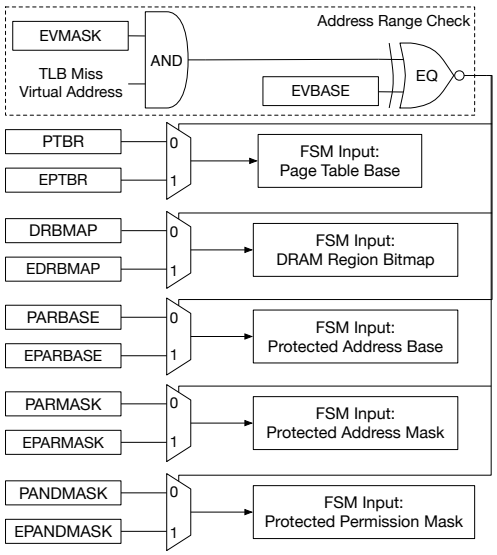

Figure 3.9: Page walker input for per-enclave page tables.

be zero), 52 AND gates, a 52-bit wide equality comparator (52 XNOR gates and 51 AND gates), and 208 (52×4) 2-bit MUXes.

3.5.3 Page Walker Memory Accesses

In modern high-speed CPUs, address translation is performed by a hardware **page walker** that traverses the page tables when a TLB miss occurs. The page walker's latency greatly impacts the CPU's performance, so it is implemented as a finite-state machine (FSM) that reads page table entries by issuing DRAM read requests using physical addresses, over a dedicated bus to the L1 cache.

Unsurprisingly, page walker modifications require a lot of engineering effort. At the same time, Sanctum's security model demands that the page walker only references enclave memory when traversing the enclave page tables, and only references OS memory when translating the OS page tables. Fortunately, these requirements can be satisfied without modifying the FSM. Instead, the security monitor configures the circuit in Figure 3.10 to ensure that the page tables only point into allowable memory.

Sanctum's security monitor must guarantee that `ptbr` points into an OS DRAM region, and `eptbr` points into a DRAM region owned by the enclave. This secures the page walker's initial DRAM read. The circuit in Figure 3.10 receives each page table entry fetched by the FSM, and sanitizes it before it reaches the page walker FSM.

The security monitor configures the set of DRAM regions that page tables may reference by writing to a DRAM region bitmap (`drbmap`) register. The sanitization circuitry extracts the DRAM region index from the address in the page table entry, and looks it up in the DRAM region bitmap. If the address does to belong to an allowable DRAM region, the sanitization logic forces the page table entry's valid bit to zero, which will cause the page walker FSM to abort the address translation and signal a page fault.

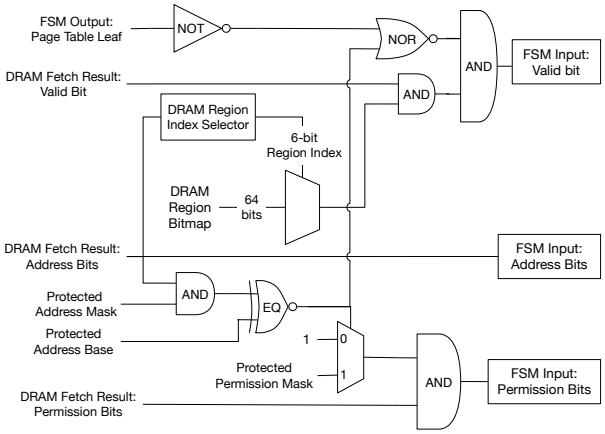

Figure 3.10: Hardware support for per-enclave page tables: check page table entries fetched by the page walker.

Sanctum's security monitor and its attestation key are stored in DRAM regions allocated to the OS. For security reasons, the OS must not be able to modify the monitor's code, or to read the attestation key. Sanctum extends the page table entry transformation described above to implement a Protected Address Range (PAR) for each set of page tables.

Each PAR is specified using a base register (`parbase`) register and a mask register (`parmask`) with the same semantics as the variable

Memory Type Range registers (MTRRs) in the x86 architecture. The page table entry sanitization logic in Sanctum's hardware extensions checks if each page table entry points into the PAR by ANDing the entry's address with the PAR mask and comparing the result with the PAR base. If a page table entry is seen with a protected address, its valid bit is cleared, forcing a page fault.

The above transformation allows the security monitor to set up a memory range that cannot be accessed by other software, and which can be used to securely store the monitor's code and data. Entry invalidation ensures no page table entries are fetched from the protected range, which prevents the page walker FSM from modifying the protected region by setting accessed and dirty bits.

All registers above are replicated, as Sanctum maintains separate OS and enclave page tables. The security monitor sets up a protected range in the OS page tables to isolate its own code and data structures (most importantly its private attestation key) from a malicious OS.

Figure 3.11 shows Sanctum's logic inserted between the page walker and the cache unit that fetches page table entries.

Assuming a 64-bit RISC-V and the example cache above, the logic requires a 64-bit MUX, 54 AND gates, a 51-bit wide equality comparator (51 XNOR gates and 50 AND gates), a 1-bit NOT gate, and a copy of the DRAM region index extraction logic in § 3.5.1, which could be just wire re-routing if the DRAM configuration is known a priori.

3.5.4 DMA Transfer Filtering

Sanctum whitelists a DMA-safe DRAM region instead of following SGX's blacklist approach. Specifically, Sanctum adds two registers (a base, `dmarbase` and an AND mask, `dmarmask`) to the DMA arbiter (memory controller). The range check circuit shown in Figure 3.9 compares each DMA transfer's start and end addresses against the allowed DRAM range, and the DMA arbiter drops transfers that fail the check.

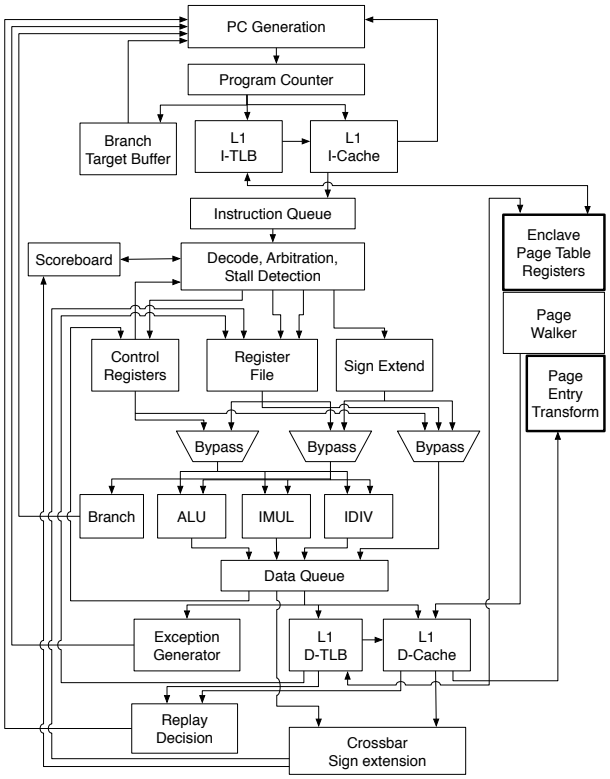

Figure 3.11: Sanctum's page entry transformation logic in the context of a Rocket core.

3.6 Software Design

Sanctum's chain of trust, discussed in § 3.6.1, diverges significantly from SGX. Sanctum replaces SGX's microcode with a software security monitor that runs at a higher privilege level than the hypervisor and the OS. On RISC-V, the security monitor runs at machine level. The Sanctum design only uses one privileged enclave, the signing enclave, which behaves similarly to SGX's Quoting Enclave.

3.6.1 Attestation Chain of Trust

Sanctum has three pieces of trusted software: the measurement root, which is burned in on-chip ROM, the security monitor (§ 3.6.2), which must be stored alongside the computer's firmware (usually in flash memory), and the signing enclave, which can be stored in any untrusted storage that the OS can access.

The expectation is that the trusted software is amenable to rigorous analysis: the implementation of a security monitor for Sanctum is written with verification in mind, and has fewer than 5 kloc of C++, including a subset of the standard library and the cryptography for enclave attestation.

The Measurement Root

The measurement root (`mroot`) is stored in a ROM at the top of the physical address space, and covers the reset vector. Its main responsibility is to compute a cryptographic hash of the security monitor and generate a monitor attestation key pair and certificate based on the monitor's hash, as shown in Figure 3.12.

The security monitor is expected to be stored in non-volatile memory (such as an SPI flash chip) that can respond to memory I/O requests from the CPU, perhaps via a special mapping in the computer's chipset. When `mroot` starts executing, it computes a cryptographic hash over the security monitor. `mroot` then reads the processor's key derivation secret, and derives a symmetric key based on the monitor's hash. `mroot` will eventually hand down the key to the monitor.

The security monitor contains a header that includes the location of an attestation key existence flag. If the flag is not set, the measurement root generates a monitor attestation key pair, and produces a monitor attestation certificate by signing the monitor's public attestation key with the processor's private attestation key. The monitor attestation certificate includes the monitor's hash.

`mroot` generates a symmetric key for the security monitor so it may encrypt its private attestation key and store it in the computer's SPI flash memory chip. When writing the key, the monitor also sets

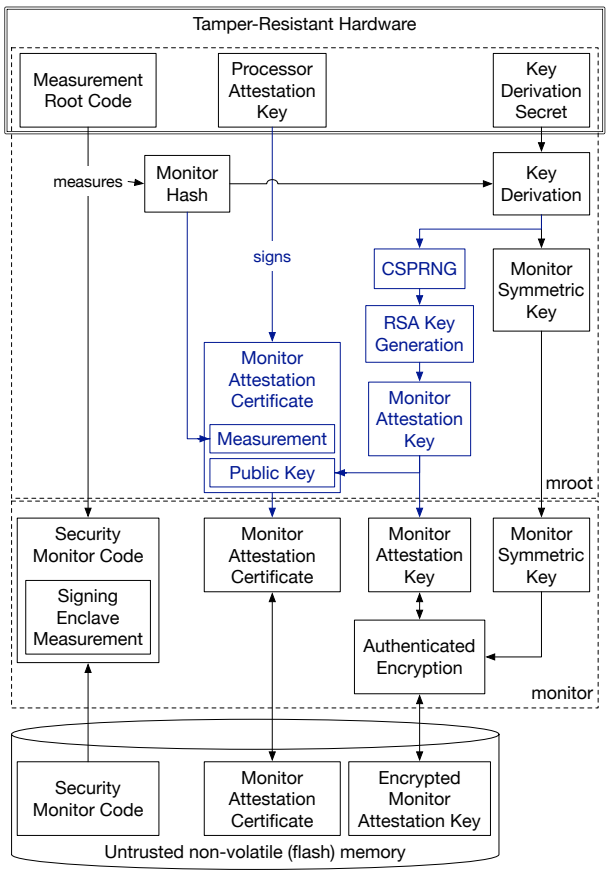

Figure 3.12: Sanctum's root of trust is a measurement root routine burned into the CPU's ROM. This code reads the security monitor from flash memory and generates an attestation key and certificate based on the monitor's hash. Asymmetric key operations, colored in blue, are only performed the first time a monitor is used on a computer.

the monitor attestation key existence flag, instructing future boot sequences not to re-generate a key. The public attestation key and certificate can be stored unencrypted in any untrusted memory.

Before handing control to the monitor, `mroot` sets a lock that blocks any software from reading the processor's symmetric key derivation seed and private key until a reset occurs. This prevents a malicious security monitor from deriving a different monitor's symmetric key, or from generating a monitor attestation certificate that includes a different monitor's measurement hash.

The symmetric key generated for the monitor is similar in concept to the Seal Keys produced by SGX's key derivation process, as it is used to securely store a secret (the monitor's attestation key) in untrusted memory, in order to avoid an expensive process (asymmetric key attestation and signing). Sanctum's key derivation process is based on the monitor's measurement, so a given monitor is guaranteed to get the same key across power cycles. The cryptographic properties of the key derivation process guarantee that a malicious monitor cannot derive the symmetric key given to another monitor.

The Signing Enclave

In order to avoid timing attacks, the security monitor does not compute attestation signatures directly. Instead, the signing algorithm is executed inside a signing enclave, which is a security monitor module that executes in an enclave environment, so it is protected by the same isolation guarantees that any other Sanctum enclave enjoys.

The signing enclave receives the monitor's private attestation key via an API call. When the security monitor receives the call, it compares the calling enclave's measurement with the known measurement of the signing enclave. Upon a successful match, the monitor copies its attestation key into enclave memory using a data-independent sequence of memory accesses, such as `memcpy`. This way, the monitor's memory access pattern does not leak the private attestation key.

Sanctum's signing enclave authenticates another enclave on the computer and securely receives its attestation data using mailboxes (§ 3.6.2), a simplified version of SGX's local attestation (report-

ing) mechanism. The enclave's measurement and attestation data are wrapped into a software attestation signature that can be examined by a remote verifier. Figure 3.13 shows the chain of certificates and signatures in an instance of software attestation.

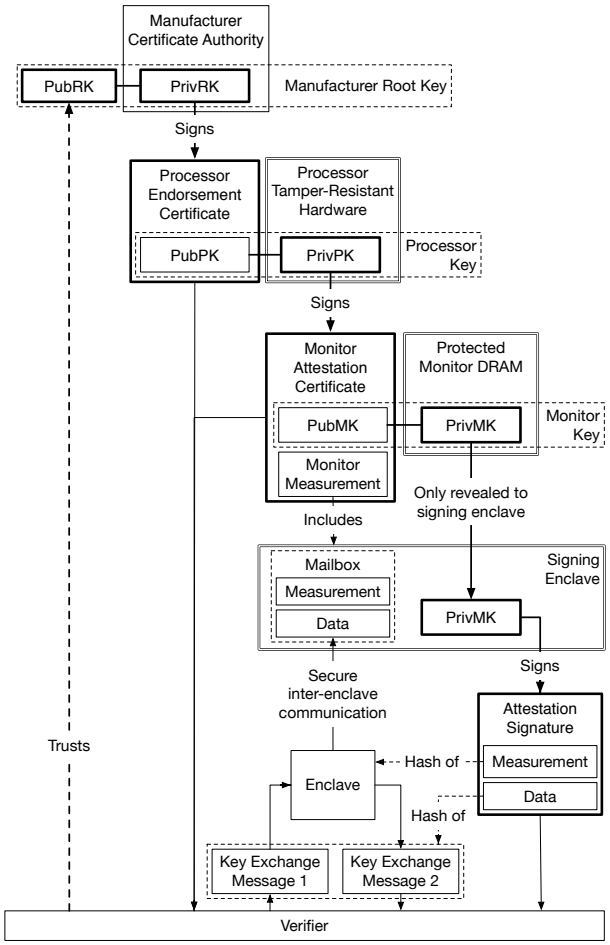

Figure 3.13: The certificate chain behind Sanctum's software attestation signatures.

3.6.2 Security Monitor

The security monitor receives control after `mroot` finishes setting up the attestation measurement chain.

The monitor provides API calls to the OS and enclaves for **DRAM region allocation** and **enclave management**. The monitor guards sensitive registers, such as the page table base register (`ptbr`) and the allowed DMA range (`dmarbase` and `dmarmask`). The OS can set these registers via monitor calls that ensure the register values are consistent with the current DRAM region allocation.

DRAM Regions

Figure 3.14 shows the DRAM region allocation state transition diagram. After the system boots up, all DRAM regions are allocated to the OS, which can free up DRAM regions so it can re-assign them to enclaves or to itself. A DRAM region can only become free after it is blocked by its owner, which can be the OS or an enclave. While a DRAM region is blocked, any address translations mapping to it cause page faults, so no new TLB entries will be created for that region. Before the OS frees the blocked region, it must flush all TLBs in order to remove any stale entries for the region.

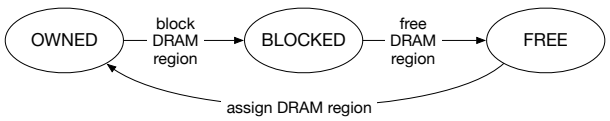

Figure 3.14: DRAM region allocation states and API calls.

The monitor ensures that the OS performs TLB shootdowns, using a global *block clock*. When a region is blocked, the block clock is incremented, and the current block clock value is stored in the metadata associated with the DRAM region (shown in Figure 3.15). When a core's TLB is flushed, that core's flush time is set to the current block clock value. When the OS asks the monitor to free a blocked DRAM region, the monitor verifies that no core's flush time is lower than the block clock value stored in the region's metadata. As an op-

timization, freeing a region owned by an enclave only requires TLB flushes on the cores running that enclave's threads. No other core can have TLB entries for the enclave's memory.

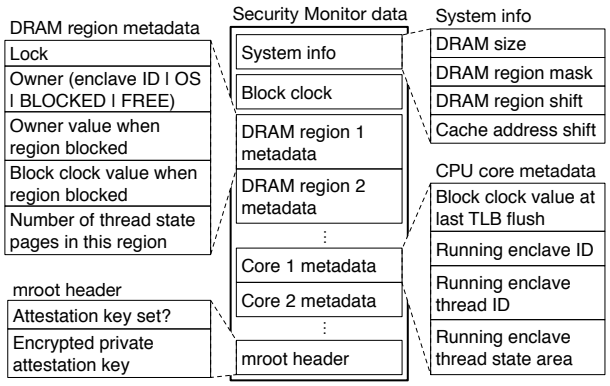

Figure 3.15: Security monitor data structures.

The region blocking mechanism guarantees that when a DRAM region is assigned to an enclave or the OS, no stale TLB mappings associated with the DRAM region exist. The monitor uses the MMU extensions described in § 3.5.2 and § 3.5.3 to ensure that once a DRAM region is assigned, no software other than the region's owner may create TLB entries pointing inside the DRAM region. Together, these mechanisms guarantee that the DRAM regions allocated to an enclave cannot be accessed by the operating system or by another enclave.

Metadata Regions

Since the security monitor sits between the OS and enclave, and its APIs can be invoked by both sides, it is an easy target for timing attacks. Sanctum prevents these attacks with a straightforward policy that states the security monitor is never allowed to access enclave data, and is not allowed to make memory accesses that depend on the attestation key material. The rest of the data handled by the monitor is derived from the OS' actions, so it is already known to the OS.

A rather obvious consequence of the policy above is that after the security monitor boots the OS, it cannot perform any crypto-

graphic operations that use keys. For example, the security monitor cannot compute an attestation signature directly, and defers that operation to a signing enclave (§ 3.6.1). While it is possible to implement some cryptographic primitives without performing data-dependent accesses, the security and correctness proofs behind these implementations are non-trivial. For this reason, Sanctum avoids depending on any such implementation.

A more subtle aspect of the access policy outlined above is that the metadata structures that the security monitor uses to operate enclaves cannot be stored in DRAM regions owned by enclaves, because that would give the OS an indirect method of accessing the LLC sets that map to enclave's DRAM regions, which could facilitate a cache timing attack.

For this reason, the security monitor requires the OS to set aside at least one DRAM region for enclave metadata before it can create enclaves. The OS has the ability to free up the metadata DRAM region, and regain the LLC sets associated with it, if it predicts that the computer's workload will not involve enclaves.

Each DRAM region that holds enclave metadata is managed independently from the other regions, at page granularity. The first few pages of each region contain a page map that that tracks the usage of each metadata page, specifically the enclave that it is assigned to, and the data structure that it holds.

Each metadata region is like an EPC region in SGX, with the exception that our metadata regions only hold special pages, like Sanctum's equivalent of SGX's Secure Enclave Control Structure (SECS) and the Thread Control Structure (TCS). These structures will be described in the following sections.

The data structures used to store Sanctum's metadata can span multiple pages. When the OS allocates such a structure in a metadata region, it must point the monitor to a sequence of free pages that belong to the same DRAM region. All the pages needed to represent the structure are allocated and released in one API call.

Enclave Lifecycle

The lifecycle of a Sanctum enclave is very similar to that of its SGX counterparts, as shown in Figure 3.16.

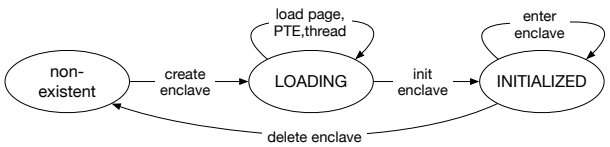

Figure 3.16: Enclave states and enclave management API calls.

The OS creates an enclave by issuing a *create enclave* call that creates the enclave metadata structure, which is Sanctum's equivalent of the SECS. The enclave metadata structure contains an array of mailboxes whose size is established at enclave creation time, so the number of pages required by the structure varies from enclave to enclave. § 3.6.2 describes the contents and use of mailboxes.

The *create enclave* API call initializes the enclave metadata fields shown in Figure 3.3, and places the enclave in the LOADING state. While the enclave is in this state, the OS sets up the enclave's initial state via monitor calls that assign DRAM regions to the enclave, create hardware threads and page table entries, and copy code and data into the enclave. The OS then issues a monitor call to transition the enclave to the INITIALIZED state, which finalizes its measurement hash. The application hosting the enclave is now free to run enclave threads.

Sanctum stores a measurement hash for each enclave in its metadata area, and updates the measurement to account for every operation performed on an enclave in the LOADING state. The policy described in § 3.6.2 does not apply to the secure hash operations used to update enclave's measurement, because the whole of the data used to compute the hash is already known to the OS.

Enclave metadata is stored in a metadata region (§ 3.6.2), so it can only be accessed by the security monitor. Therefore, the metadata area can safely store public information with integrity requirements, such as the enclave's measurement hash.

While an OS loads an enclave, it is free to map the enclave's pages, but the monitor maintains its page tables ensuring all entries point to non-overlapping pages in DRAM owned by the enclave. Once an enclave is initialized, it can inspect its own page tables and abort if the OS created undesirable mappings. Simple enclaves do not require specific mappings. Complex enclaves are expected to communicate their desired mappings to the OS via out-of-band metadata not covered by this work.

Sanctum's monitor ensures that page tables do not overlap by storing the last mapped page's physical address in the enclave's metadata. To simplify the monitor, a new mapping is allowed if its physical address is greater than that of the last, constraining the OS to map an enclave's DRAM pages in monotonically increasing order.

Enclave Code Execution

Sanctum closely follows the threading model of SGX enclaves. Each CPU core that executes enclave code uses a thread metadata structure, which is Sanctum's equivalent of SGX's TCS combined with SGX's State Save Area (SSA). Thread metadata structures are stored in a DRAM region dedicated to enclave metadata in order to prevent a malicious OS from mounting timing attacks against an enclave by causing AEXes on its threads. Figure 3.17 shows the lifecycle of a thread metadata structure.

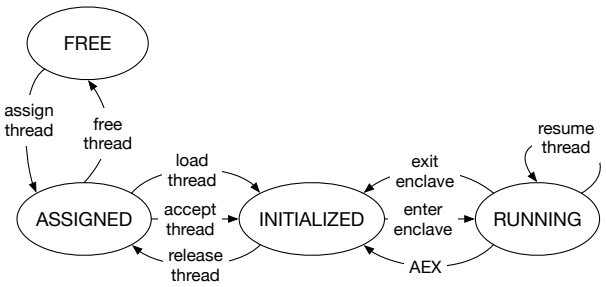

Figure 3.17: Enclave thread metadata structure states and thread-related API calls.

The OS turns a sequence of free pages in a metadata region into an uninitialized thread structure via an *allocate thread* monitor call. Dur-

ing enclave loading, the OS uses a *load thread* monitor call to initialize the thread structure with data that contributes to the enclave's measurement. After an enclave is initialized, it can use an *accept thread* monitor call to initialize its thread structure.

The application hosting an enclave starts executing enclave code by issuing an *enclave enter* API call, which must specify an initialized thread structure. The monitor honors this call by configuring Sanctum's hardware extensions to allow access to the enclave's memory, and then by loading the program counter and stack pointer registers from the thread's metadata structure. The enclave's code can return control to the hosting application voluntarily, by issuing an *enclave exit* API call, which restores the application's PC and SP from the thread state area and sets the API call's return value to `ok`.

When performing an AEX, the security monitor atomically tests-and-sets the *AEX state valid* flag in the current thread's metadata. If the flag is clear, the monitor stores the core's execution state in the thread state's AEX area. Otherwise, the enclave thread was resuming from an AEX, so the monitor does not change the AEX area. When the host application re-enters the enclave, it will resume from the previous AEX. This reasoning avoids the complexity of SGX's state stack.

If an interrupt occurs while the enclave code is executing, the security monitor's exception handler performs an AEX, which sets the API call's return value to `async_exit`, and invokes the standard interrupt handling code. After the OS handles the interrupt, the enclave's host application resumes execution, and re-executes the *enter enclave* API call. The enclave's thread initialization code examines the saved thread state, and seeing that the thread has undergone an AEX, issues a *resume thread* API call. The security monitor restores the enclave's registers from the thread state area, and clears the AEX flag.

Mailboxes

Sanctum's software attestation process relies on *mailboxes*, which are a simplified version of SGX's local attestation mechanism. Sanctum could not follow SGX's approach because it relies on key derivation and MAC algorithms, and Sanctum's timing attack avoidance pol-

icy (§ 3.6.2) states that the security monitor is not allowed to perform cryptographic operations that use keys.

Each enclave's metadata area contains an array of mailboxes, whose size is specified at enclave creation time, and covered by the enclave's measurement. Each mailbox goes through the lifecycle shown in Figure 3.18.

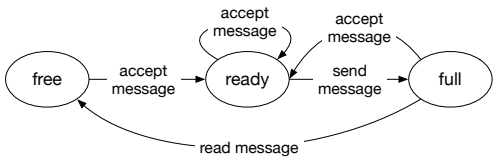

Figure 3.18: Mailbox states and security monitor API calls related to inter-enclave communication.

An enclave that wishes to receive a message in a mailbox, such as the signing enclave, declares its intent by performing an *accept message* monitor call. The API call is used to specify the mailbox that will receive the message, and the identity of the enclave that is expected to send the message.

The sending enclave, which is usually the enclave wishing to be authenticated, performs a *send message* call that specifies the identity of the receiving enclave, and a mailbox within that enclave. The monitor only delivers messages to mailboxes that expect them. At enclave initialization, the expected sender for all mailboxes is an invalid value (all zeros), so the enclave will not receive messages until it calls *accept message*.

When the receiving enclave is notified via an out-of-band mechanism that it has received a message, it issues a *read message* call to the monitor, which moves the message from the mailbox into the enclave's memory. If the API call succeeds, the receiving enclave is assured that the message was sent by the enclave whose identity was specified in the *accept message* call.

Enclave mailboxes are stored in metadata regions (§ 3.6.2), which cannot be accessed by any software other than the security monitor.

This guarantees the privacy, integrity, and freshness of the messages sent via the mailbox system.

Sanctum's mailbox design has the downside that both the sending and receiving enclave need to be alive in DRAM in order to communicate. By comparison, SGX's local attestation can be done asynchronously. In return, mailboxes do not require any cryptographic operations, and have a much simpler correctness argument.

Multi-Core Concurrency

The security monitor is highly concurrent, with fine-grained locks. API calls targeting two different enclaves may be executed in parallel on different cores. Each DRAM region has a lock guarding that region's metadata. An enclave is guarded by the lock of the DRAM region holding its metadata. Each thread metadata structure also has a lock guarding it, which is acquired when the structure is accessed, but also while the metadata structure is used by a core running enclave code. Thus, the *enter enclave* call acquires a slot lock, which is released by an *enclave exit* call or by an AEX.

Sanctum avoids deadlocks by using a form of optimistic locking. Each monitor call attempts to acquire all necessary locks via atomic test-and-set operations, and errors with a `concurrent_call` code if any lock is unavailable.

3.6.3 Enclave Eviction

General-purpose software can be enclaved without source code changes, provided that it is linked against a runtime (e.g., *libc*) modified to work with Sanctum. Any such runtime would be included in the TCB.

The Sanctum design allows the operating system to over-commit physical memory allocated to enclaves, by collaborating with an enclave to page some of its DRAM regions to disk. Sanctum does not give the OS visibility into enclave memory accesses, in order to prevent private information leaks, so the OS must decide the enclave whose DRAM regions will be evicted based on other activity, such as network I/O, or based on a business policy, such as Amazon EC2's spot instances.

Once a victim enclave has been decided, the OS asks the enclave to block a DRAM region (cf. Figure 3.14), giving the enclave an opportunity to rearrange data in its RAM regions. DRAM region management can be transparent to the programmer if handled by the enclave's runtime. The presented design requires each enclave to always occupy at least one DRAM region, which contains enclave data structures and the memory management code described above. Evicting all of a live enclave's memory requires an entirely different scheme that is deferred to future work.

The security monitor does not allow the OS to forcibly reclaim a single DRAM region from an enclave, as doing so would leak memory access patterns. Instead, the OS can delete an enclave, after stopping its threads, and reclaim all its DRAM regions. Thus, a small or short-running enclave may well refuse DRAM region management requests from the OS, and expect the OS to delete and restart it under memory pressure.

To avoid wasted work, large long-running enclaves may elect to use demand paging to overcommit their DRAM, albeit with the understanding that demand paging leaks page-level access patterns to the OS. Securing this mechanism requires the enclave to obfuscate its page faults via periodic I/O using oblivious RAM techniques, as in the Ascend processor [Fletcher et al., 2012], applied at page rather than cache line granularity, and with integrity verification. This carries a high overhead: even with a small chance of paging, an enclave must generate periodic page faults, and access a large set of pages at each period. Using an analytic model, we estimate the overhead to be upwards of 12ms per page per period for a high-end 10K RPM drive, and 27ms for a value hard drive. Given the number of pages accessed every period grows with an enclave's data size, the costs are easily prohibitive. While SSDs may alleviate some of this prohibitive overhead, and will be investigated in future work, currently Sanctum focuses on securing enclaves without demand paging.

Enclaves that perform other data-dependent communication, such as targeted I/O into a large database file, must also use the periodic oblivious I/O to obfuscate their access patterns from the operating

system. These techniques are independent of application business logic, and can be provided by libraries such as database access drivers.

Briefly, the OS can ask the security monitor to *freeze* an enclave, encrypting the enclave's DRAM regions in place, and creating a leaf node in a hash tree. When the monitor *thaws* a frozen enclave, it uses the hash tree leaf to ensure freshness, decrypts the DRAM regions, and *relocates* the enclave, updating its page tables to reflect new owners of relevant DRAM regions. The hash tree is managed by the OS using an approach similar to SGX's version array page eviction.

3.7 Security Analysis of Sanctum

Sanctum's security argument rests on two pillars: the enclave isolation enforced by the security monitor, and the guarantees behind the software attestation signature. This section outlines correctness arguments for each of these pillars.

Sanctum's isolation primitives protect enclaves from outside software that attempts to observe or interact with the enclave software via means outside the interface provided by the security monitor. Sanctum prevents direct attacks by ensuring that the memory owned by an enclave can only be accessed by that enclave's software. More subtle attacks are foiled by also isolating the structures used to access the enclave's memory, such as the enclave's page tables and the caches that hold enclave data, and by sanitizing hardware structures updated by enclave execution, such as the cores' branch prediction tables. Attacks exploiting DMA-capable devices on the system bus are thwarted by whitelisting a region of physical addresses where DMA accesses are permitted, and the DRAM controller timing channel is closed by either expecting the OS to exclusively grant DRAM banks to enclaves (this is evident during attestation), or by implementing a DRAM controller with a constant expected access latency.

3.7.1 Protection Against Direct Attacks

The correctness proof for Sanctum's DRAM isolation can be divided into two sub-proofs that cover the hardware and software sides of the

system. First, we need to prove that the page walker modifications described in § 3.5.2 and § 3.5.3 behave according to their descriptions. Thanks to the small sizes of the circuits involved, this sub-proof can be accomplished by simulating the circuits for all logically distinct input cases. Second, we must prove that the security monitor configures Sanctum's extended page walker registers in a way that prevents direct attacks on enclaves. This part of the proof is significantly more complex, but it follows the same outline as the proof for SGX's memory access protection presented in § 2.3.

The proof revolves around a main invariant stating that all TLB entries in every core are consistent with the programming model described in § 3.2. The invariant breaks down into three cases that match § 2.3, after substituting DRAM regions for pages.

These three cases are outlined below.

1. At all times when a core is not executing enclave code, its TLB may only contain physical addresses in DRAM regions allocated to the OS.

2. At all times when a core is executing an enclave's code, the TLB entries for virtual addresses outside the current enclave's EVRANGE must contain physical addresses belonging to DRAM regions allocated to the OS.

3. At all times when an core is executing an enclave's code, the TLB entries for virtual addresses inside the current enclave's EVRANGE must match the virtual memory layout specified by the enclave's page tables.

3.7.2 Protection Against Subtle Attacks

Sanctum also protects enclaves from software attacks that attempt to exploit side channels to obtain information indirectly. We focus on proving that Sanctum protects against the attacks mentioned in § 3.8, which target the page fault address and cache timing side-channels.

The proof that Sanctum foils page fault attacks is centered around the claims that each enclave's page fault handler and page tables and

page fault handler are isolated from all other software entities on the computer. First, all the page faults inside an enclave's EVRANGE are reported to the enclave's fault handler, so the OS cannot observe the virtual addresses associated with the faults. Second, page table isolation implies that the OS cannot access an enclave's page tables and read the access and dirty bits to learn memory access patterns.

Page table isolation is a direct consequence of the claim that Sanctum correctly protects enclaves against direct attacks, which was covered above. Each enclave's page tables are stored in DRAM regions allocated to the enclave, so no software outside the enclave can access these page tables.

The proof behind Sanctum's cache isolation is straightforward but tedious, as there are many aspects involved. We start by peeling off the easier cases, and tackle the most difficult step of the proof at the end of the section. Sanctum assumes the presence of both per-core caches and a shared LLC, and each cache type requires a separate correctness argument. Per-core cache isolation is achieved simply by flushing core-private caches at every transition between enclave and non-enclave mode. The cores' microarchitectural state containing private information — TLB (§ 3.7.1), and branch history table — is likewise sanitized by the security monitor. DRAM controller timing is sanitized by implementing a controller with constant expected access latency (or by refusing trust in enclaves that aren't granted DRAM regions covering entire DRAM banks). To prove the correctness of LLC isolation, we first show that enclaves do not share LLC lines with outside software, and then we show that the OS cannot indirectly reach into an enclave's LLC lines via the security monitor.

The two invariants outlined below show that per-core caches are never shared between an enclave and any other software, effectively proving the correctness of per-core cache isolation.

1. At all times when a core is not executing enclave code, its private caches only contain data accessed by the OS (and other non-enclave software).

2. At all times when a core is executing enclave code, its private caches only contain data accessed by the currently running enclave.

Due to the private nature of the per-core caches, the two invariants can be proven independently for every core, by induction over the sequence of instructions executed by the core. The base case occurs when the security monitor hands off the computer's control to the OS, at which point there is no enclave, so all caches contain OS data. The induction step has two cases – transitions between the OS and an enclave flush the per-core caches, while all the other instructions trivially follow the invariants.

Showing that enclaves do not share LLC lines with outside software can be accomplished by proving a stronger invariant that states at all times, any LLC line that can potentially cache a location in an enclave's memory cannot cache any location outside that enclave's memory. In steady state, this follows directly from the LLC isolation scheme in § 3.5.1, because the security monitor guarantees that each DRAM region is assigned to exactly one enclave or to the OS.

The situations where a DRAM region changes ownership, outlined below, can be reasoned case by case.

1. DRAM regions allocated to un-initialized enclaves can be indirectly accessed by the OS, using the monitor APIs for loading enclaves. It follows that the OS can influence the state of the enclave's LLC lines at the moment when the enclave starts. Each enclave's initialization code is expected to read enough memory in each DRAM region to get the LLC lines assigned to the enclave's memory in a known state. This way, the OS cannot influence the timing of the enclave's memory accesses, by adjusting the way it loads the enclave's pages.

2. When an enclave starts using a DRAM region allocated by the OS after the enclave was initialized, it must follow the same process outlined above to drive the LLC lines mapping the DRAM region into a pre-determined state. Otherwise, the OS can influence the timing of the enclave's memory accesses, as reasoned above.

3. When an enclave blocks a DRAM region in order to free it for OS use, the enclave is responsible for cleaning up the DRAM region's contents and getting the associated LLC lines in a state that does not leak secrets. Enclave runtimes that implement DRAM region management can accomplish both tasks by zeroing the DRAM region before blocking it.

4. DRAM regions released when the OS terminates an enclave are zeroed by the security monitor. This removes secret data from the DRAM regions, and also places the associated LLC lines in a predetermined state. Thus, the OS cannot use probing to learn about the state of the LLC lines that mapped the enclave's DRAM regions.

The implementation for the measures outlined above belongs in the enclave runtime (e.g., a modified libc), so enclave application developers do not need to be aware of this security argument.

Last, we focus on the security monitor, because it is the only piece of software outside an enclave that can access the enclave's DRAM regions. In order to claim that an enclave's LLC lines are isolated from outside software, we must prove that the OS cannot use the security monitor's API to indirectly modify the state of the enclave's LLC lines. This proof is accomplished by considering each function exposed by the monitor API, as well as the monitor's hardware fault handler. The latter is considered to be under OS control because in a worst case scenario, a malicious OS could program peripherals to cause interrupts as needed to mount a cache timing attack.

First, we ignore all API functions that do not directly operate on enclave memory, such as the APIs that manage DRAM regions. The ignored functions include many APIs that manage enclave-related structures, such as mailbox APIs (§ 3.6.2) and most thread APIs (§ 3.6.2), because they only operate on enclave metadata stored in dedicated metadata DRAM regions. In fact, the sole purpose of metadata DRAM regions is being able to ignore most APIs in this security argument.

Second, we ignore the API calls that operate on un-initialized enclaves, because each enclave is expected to drive its LLC lines into a known state during initialization.

The remaining API call is *enter enclave*, which causes the current core to start executing enclave code. All enclave code must be stored in the enclave's DRAM regions, so it can receive integrity guarantees. It follows that the OS can use *enter enclave* to cause specific enclave LLC lines to be fetched. This API does not contradict our security argument because *enter enclave* is the Sanctum-provided method for outside software to call into the enclave, so it is effectively a part of the enclave's interface.

We note that enclaves have means to control *enter enclave*'s behavior. The API call will only access enclave memory if the thread metadata (§ 3.6.2) passed to it is available. Furthermore, code execution starts at an entry point defined by the enclave.

Last, we analyze the security monitor's handling of hardware exceptions. We discuss faults (such as division by zero and page faults) and interrupts caused by peripherals separately, as they are handled differently.

When a core starts executing enclave code, the security monitor configures it to route faults to an enclave-defined handler. On RISC-V, this is done without executing any non-enclave code. On architectures where the fault handler is always invoked with monitor privileges, the security monitor must copy its fault handler inside each enclave's DRAM regions, and configure the `eparbase` and `eparmask` registers (§ 3.5.3) to prevent the enclave from modifying the handler code. Faults must not be handled by the security monitor code stored in OS DRAM regions, because that would give a malicious OS an opportunity to learn when enclaves incur faults, via cache probing.

Sanctum's security monitor handles interrupts received during enclave execution by performing an AEX (§ 3.2). The AEX implementation only accesses information in metadata DRAM regions, and writes the core's execution state to a metadata DRAM region. Since the AEX does not access the enclave's DRAM regions, its code can be safely stored in OS DRAM regions, along with the rest of the security mon-

itor. The OS can observe AEXes by probing the LLC lines storing the AEX handler. However, this leaks no information, because each AEX results OS code execution.

3.7.3 Operating System Protection

Sanctum protects the operating system from direct attacks against malicious enclaves, but does not protect it against subtle attacks that take advantage of side-channels. Sanctum assumes that software developers will transition all sensitive software into enclaves, which are protected even if the OS is compromised. At the same time, a honest OS can potentially take advantage of Sanctum's DRAM regions to isolate mutually mistrusting processes.

Proving that a malicious enclave cannot attack the host computer's operating system is accomplished by first proving that the security monitor's APIs that start executing enclave code always place the core in unprivileged mode, and then proving that the enclave can only access OS memory using the OS-provided page tables. The first claim can be proven by inspecting the security monitor's code. The second claim follows from the correctness proof of the circuits in § 3.5.2 and § 3.5.3. Specifically, each enclave can only access memory either via its own page tables or the OS page tables, and the enclave's page tables cannot point into the DRAM regions owned by the OS.

These two claims effectively show that Sanctum enclaves run with the privileges of their host application. This parallels SGX, so all arguments about OS security in SGX apply to Sanctum as well. Specifically, malicious enclaves cannot DoS the OS, and can be contained using the mechanisms that currently guard against malicious user software.

3.7.4 Security Monitor Protection

The security monitor is in Sanctum's TCB, so the system's security depends on the monitor's ability to preserve its integrity and protect its secrets from attackers. The monitor does not use address translation, so it is not exposed to any attacks via page tables. The monitor also does not protect itself from cache timing attacks, and instead avoids making any memory accesses that would reveal sensitive information.

Proving that the monitor is protected from direct attacks from a malicious OS or enclave can be accomplished in a few steps. First, we invoke the proof that the circuits in § 3.5.2 and § 3.5.3, are correct. Second, we must prove that the security monitor configures Sanctum's extended page walker registers correctly. Third, we must prove that the DRAM regions that contain monitor code or data are always allocated to the OS.

The circuit correctness proof was outlined in § 3.7.1, and effectively comes down to reasoning through all possible input classes and simulating the circuit.

The register configuration correctness proof consists of analyzing Sanctum's initialization code and ensuring that it sets up the `parbase` and `parmask` registers to cover all of the monitor's code and data. These registers will not change after the security monitor hands off control to the OS in the boot sequence.

Last, proving that the DRAM regions that store the monitor always belong to the OS requires analyzing the monitor's DRAM region management code. At boot time, all DRAM regions must be allocated to the OS correctly. During normal operation, the *block DRAM region* API call (§ 3.6.2) must reject DRAM regions that contain monitor code or data. At a higher level, the DRAM region management code must implement the state machine shown in Figure 3.14, and the `drbmap` and `edrbmap` registers (§ 3.5.3) must be updated to correctly reflect DRAM region ownership.

Since the monitor is exposed to cache timing attacks from the OS, Sanctum's security guarantees rely on proofs that the attacks would not yield any information that the OS does not already have. Fortunately, most of the security monitor implementation consists of acknowledging and verifying the OS' resource allocation decisions. The main piece of private information held by the security monitor is the attestation key. We can be assured that the monitor does not leak this key, as long as we can prove that the monitor implementation only accesses the key when it is provided to the signing enclave (§ 3.6.1), that the key is provided via a data-independent memory copy operation, such as `memcpy`, and that the attestation key is only disclosed to the signing enclave.

Superficially, proving the last claim described above comes down to ensuring that the API providing the key compares the current enclave's measurement with a hard-coded value representing the correct signing enclave, and errors out in case of a mismatch. However, the current enclave's measurement is produced by a sequence of calls to the monitor's enclave loading APIs, so a complete proof also requires analyzing each loading API implementation and proving that it modifies the enclave measurement as expected.

Sanctum's monitor requires a complex security argument when compared to SGX's microcode, because the microcode is burned into a ROM that is not accessible by software, and is connected to the execution core via a special path that bypasses caches. We expect that the extra complexity in the security argument is much smaller than the complexity associated with the microcode programming. Furthermore, SGX's architectural enclaves, such as its quoting enclave, must operate under the same regime as Sanctum's monitor, as SGX does not guarantee cache isolation to its enclaves.

3.7.5 The Security of Software Attestation

The security of Sanctum's software attestation scheme depends on the correctness of the measurement root and the security monitor. `mroot`'s sole purpose is to set up the attestation chain, so the attestation's security requires the correctness of the entire `mroot` code. The monitor's enclave measurement code also plays an essential role in the attestation process, because it establishes the identity of the attested enclaves, and is also used to distinguish between the signing enclave and other enclaves. Sanctum's attestation also relies on mailboxes, which are used to securely transmit attestation data from the attested enclave to the signing enclave.

At a high level, the measurement root's correctness proof is lengthy and tedious, because setting up the attestation chain requires implementations for cryptographic hashing, symmetric encryption, RSA key generation, and RSA signing. Before the monitor measurement is performed, the processor must be placed into a cache-as-RAM mode, and the monitor must be copied into the processor's cache. Fortunately, the

proof can be simplified by taking into account that when `mroot` starts executing, the computer is in a well-defined post-reset state, interrupt processing is disabled, and no other software is being executed by the CPU. Furthermore, `mroot` runs in machine mode on the RISC-V architecture, so the code uses physical addresses, and does not have to deal with the complexities of address translation.

Measuring the security monitor requires initializing the flash memory that stores it. Interestingly, the correctness of the initialization code is not critical to Sanctum's security. If the flash memory is set up incorrectly, the monitor software that is executed may not match the version stored on the flash chip. However, this does not impact the software attestation's security, as long as the measurement computed by `mroot` matches the monitor that is executed. Storage initialization code is generally complex, so this argument can be used to exclude a non-trivial piece of the measurement root's code from correctness proofs.

The correctness proof for the monitor's measurement code consists of a sub-proof for the cryptographic hash implementation, and sub-proofs for the code that invokes the cryptographic hash module in each of the enclave loading APIs (§ 3.6.2), which are *create enclave*, *load page*, *load page table entry*, and *load thread*. The cryptographic hash implementation can be shared with the measurement root, so the correctness proof can be shared as well. The hash implementation operates on public data, so it does not need to be resistant to side-channel attacks. The correctness proofs for the enclave loading API implementations are tedious but straightforward.

The security analysis of the monitor's mailbox API (§ 3.6.2) implementation relies on proving the claims below. Each claim can be proved by analyzing a specific part of the mailbox module's code. It is worth noting that these claims only cover the subset of the mailbox implementation that the signing enclave depends on.

1. An enclave's mailboxes are initialized to the empty state.

2. The *accept message* API correctly sets the target mailbox into the empty state, and copies the expected sender enclave's measurement into the mailbox metadata.

3. The *send message* API errors out without making any modifications if the sending enclave's measurement does not match the expected value in the mailbox's metadata.

4. The *send message* API copies the sender's message into the correct field of the mailbox's metadata.

5. The *receive message* API errors out if the mailbox is not in the full state.

6. The *receive message* API correctly copies the message from the mailbox metadata to the calling enclave.

3.8 Work Related to Sanctum Mechanisms

Sanctum's main improvement over SGX is preventing software attacks that analyze an isolated container's memory access patterns to infer private information. One should be concerned with cache timing attacks [Banescu, 2011], because they can be mounted by unprivileged software sharing a computer with the victim software.

Cache timing attacks are known to retrieve cryptographic keys used by AES [Bonneau and Mironov, 2006], RSA [Brumley and Boneh, 2005], Diffie-Hellman [Kocher, 1996], and elliptic-curve cryptography [Brumley and Tuveri, 2011]. While early attacks required access to the victim's CPU core, recent sophisticated attacks [Yarom and Falkner, 2013, Liu et al., 2015] target the last-level cache (LLC), which is shared by all cores in a socket. Recently, [Oren et al., 2015] demonstrated a cache timing attack that uses JavaScript code in a page visited by a web browser.

Cache timing attacks observe a victim's memory access patterns at cache line granularity. However, recent work shows that private information can be gleaned even from the page-level memory access pattern obtained by a malicious OS that simply logs the addresses seen by its page fault handler [Xu et al., 2015].

The research community has brought forward various defenses against cache timing attacks. PLcache [Wang and Lee, 2007, Kong et al., 2008] and the Random Fill Cache Architecture (RFill, [Liu and

Lee, 2014]) were designed and analyzed in the context of a small region of sensitive data, and scaling them to protect a potentially large enclave without compromising performance is not straightforward. When used to isolate entire enclaves in the LLC, RFill performs at least 37%-66% worse than Sanctum.

RPcache [Wang and Lee, 2007, Kong et al., 2008] trusts the OS to assign different hardware process IDs to mutually mistrusting entities, and its mechanism does not directly scale to large LLCs. The non-monopolizable cache [Domnitser et al., 2012] uses a well-principled partitioning scheme, but does not completely stop leakage, and relies on the OS to assign hardware process IDs. CATalyst [Liu et al., 2016] trusts the Xen hypervisor to correctly tame Intel's Cache Allocation Technology into providing cache pinning, which can only secure software whose code and data fits into a fraction of the LLC.

A fair share of the cache timing attack countermeasures cited here focus on protecting relatively small pieces of code and data that are loosely coupled to the rest of the application. The countermeasures are suitable for cryptographic keys and the algorithms that operate on them, but do not scale to larger codebases. This is a questionable approach, because crypto keys have no intrinsic value, and are only attacked to gain access to the sensitive data that they protect. For example, in a medical image processing application, the sensitive data may be patient X-rays. A high-resolution image uses at least a few megabytes, so the countermeasures above will leave the X-rays vulnerable to cache timing attacks while they are operated on by image processing algorithms.

Sanctum uses very simple cache partitioning [Lin et al., 2008] based on page coloring [Taylor et al., 1990, Kessler and Hill, 1992], which has proven to have reasonable overheads. It is likely that sophisticated schemes like ZCache [Sanchez and Kozyrakis, 2010] and Vantage [Sanchez and Kozyrakis, 2011] can be combined with Sanctum's framework to yield better performance.

4

Conclusion

This manuscript is the second of a two-part review of secure processor systems that aims to enable remote computation with guarantees of privacy and integrity. Part I of this survey established the background, taxonomy, and prior work relevant for a discussion of trusted remote computation. Part I also described the secure isolation container (enclave), as exemplified by Intel's Secure Guard Extensions (SGX), as an effective primitive for secure remote computation. This manuscript, Part II, relies on the fundamentals discussed previously and extends the discussion of enclaves with an in-depth investigation of the implementation, security-critical mechanisms, threat model, and shortcomings of SGX.

Shortly after we learned about Intel's Software Guard Extensions (SGX) initiative, we set out to study it in the hope of finding a practical solution to its vulnerability to cache timing attacks. After reading the official SGX manuals, we were left with more questions than when we started. The SGX patents filled some of the gaps in the official documentation, but also revealed Intel's enclave licensing scheme, which has troubling implications.

After learning about the SGX implementation and inferring its design constraints, we discarded our draft proposals for defending enclave software against cache timing attacks. We concluded that it would be impossible to claim to provide this kind of guarantee given the design constraints and all unknowns surrounding the SGX implementation. Instead, we applied the knowledge that we gained to design the MIT Sanctum processor.

Sanctum shows that strong provable isolation of concurrent software modules can be achieved with low design complexity and little overhead. This approach provides strong security guarantees against an insidious software threat model including cache timing and memory access pattern attacks. With the design of Sanctum, we hope to enable a shift in discourse in secure hardware architecture away from plugging specific security holes to a principled approach to eliminating attack surfaces.

This two part survey and analysis describes our findings while studying SGX and designing Sanctum. We hope that it will help fellow researchers understand the breadth of issues that need to be considered before accepting a trusted hardware design into a trusted computing base. We also hope that our work will prompt the research community to demand more transparency and open design from the vendors who ask us to trust their hardware.

Acknowledgments

Funding for this research was partially provided by the National Science Foundation under contract number CNS-1413920 and by Delta Electronics. We thank Christopher Fletcher, Albert Kwon, Marten van Dijk, Ling Ren, Ron Rivest, and Nickolai Zeldovich for useful discussions throughout the course of this work. We acknowledge the useful feedback from Intel SGX designers on an early version of this manuscript.

References

Linux kernel: CVE security vulnerabilities, versions and detailed reports. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33, 2014a. [Online; accessed 27-April-2015].

XEN: CVE security vulnerabilities, versions and detailed reports. http://www.cvedetails.com/product/23463/XEN-XEN.html?vendor_id=6276, 2014b. [Online; accessed 27-April-2015].

Xen project software overview. http://wiki.xen.org/wiki/Xen_Project_Software_Overview, 2015. [Online; accessed 27-April-2015].

Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. 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, HASP*, volume 13, 2013.

Sebastian Anthony. Who actually develops Linux? the answer might surprise you. http://www.extremetech.com/computing/175919-who-actually-develops-linux, 2014. [Online; accessed 27-April-2015].

Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-VM attacks on Xen and VMware are possible! Cryptology ePrint Archive, Report 2014/248, 2014. http://eprint.iacr.org/.

Sebastian Banescu. Cache timing attacks. 2011. [Online; accessed 26-January-2014].

Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 201–215. Springer, 2006.

E. Brickell and J. Li. Hardening inter-device secure communication using physically unclonable functions, 2014. US Patent App. 13/844,559.

Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing. *IACR Cryptology ePrint Archive*, 2009.

Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*, pages 355–371. Springer, 2011.

David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. In *Proceedings of the 9th annual ACM Symposium on Theory of Computing*, pages 106–112. ACM, 1977.

Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

Victor Costan and Srinivas Devadas. Security challenges and opportunities in adaptive and reconfigurable hardware. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 1–5. IEEE, 2011.

Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, Feb 2016.

Victor Costan, Ilia Lebedev, and Srinivas Devadas. Secure processors part I: Background, taxonomy for secure enclaves and Intel SGX architecture. In *FnTEDA*, 2017.

Shaun Davenport. SGX: the good, the bad and the downright ugly. *Virus Bulletin*, 2014.

Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

Loïc Duflot, Daniel Etiemble, and Olivier Grumelard. Using CPU system management mode to circumvent operating system security functions. *CanSecWest/core06*, 2006.

Alan Dunn, Owen Hofmann, Brent Waters, and Emmett Witchel. Cloaking malware with the trusted platform module. In *USENIX Security Symposium*, 2011.

Shawn Embleton, Sherri Sparks, and Cliff C. Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 2010.

Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-X: A flexible architecture for hardware-managed isolated execution. In *Microarchitecture (MICRO), 2014 47th annual IEEE/ACM International Symposium on*, pages 190–202. IEEE, 2014.

Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, pages 3–8. ACM, 2012.

Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 148–160. ACM, 2002.

Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the 19th annual ACM symposium on Theory of Computing*, pages 182–194. ACM, 1987.

K. C. Gotze, G. M. Iovino, and J. Li. Secure provisioning of secret keys during integrated circuit manufacturing, 2014a. US Patent App. 13/631,512.

K. C. Gotze, J. Li, and G. M. Iovino. Fuse attestation to secure the provisioning of secret keys during integrated circuit manufacturing, 2014b. US Patent 8,885,819.

David Grawrock. *Dynamics of a Trusted Platform: A building block approach.* Intel Press, 2009.

Shay Gueron. Quick verification of RSA signatures. In *8th International Conference on Information Technology: New Generations (ITNG)*, pages 382–386. IEEE, 2011.

Shay Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016.

Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, volume 13, 2013.

Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-VM RSA key recovery in a public cloud. Cryptology ePrint Archive, Report 2015/898, 2015.

Software Guard Extensions Programming Reference. Intel Corporation, 2013. Reference no. 329298-001US.

Software Guard Extensions Programming Reference. Intel Corporation, 2014. Reference no. 329298-002US.

Intel® Software Guard Extensions (Intel® SGX). Intel Corporation, Jun 2015a. Reference no. 332680-002.

Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation, Sep 2015b. Reference no. 325462-056US.

Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® software guard extensions: EPID provisioning and attestation services. `https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services`, Mar 2016. [Online; accessed 21-Mar-2016].

Simon P. Johnson, Uday R. Savagaonkar, Vincent R. Scarlata, Francis X. McKeen, and Carlos V. Rozas. Technique for supporting multiple secure enclaves, Dec 2010. US Patent 8,972,746.

Richard E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10 (4):338–359, 1992.

Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceeding of the 41st annual International Symposium on Computer Architecuture*, pages 361–372. IEEE Press, 2014.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology–CRYPTO'96*, pages 104–113. Springer, 1996.

Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on Computer security architectures*, pages 25–34. ACM, 2008.

Tsvika Kurts, Guillermo Savransky, Jason Ratner, Eilon Hazan, Daniel Skaba, Sharon Elmosnino, and Geeyarpuram N. Santhanakrishnan. Generic debug eXternal connection (GDXC) for high integration integrated circuits, 2011. US Patent 8,074,131.

Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *CoRR*, abs/1611.06952, 2016. URL `http://arxiv.org/abs/1611.06952`.

Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanovic, and Krste Asanovic. A 45nm 1.3 GHz 16.7 double-precision GFLOPS/w RISC-V processor with vector accelerators. In *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*, pages 199–202. IEEE, 2014.

Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *14th International IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 367–378. IEEE, 2008.

Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Microarchitecture (MICRO), 2014 47th annual IEEE/ACM International Symposium on*, pages 203–215. IEEE, 2014.

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 143–158. IEEE, 2015.

Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, Mar 2016.

R. Maes, P. Tuyls, and I. Verbauwhede. Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 332–347, 2009.

Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

Francis X. McKeen, Carlos V. Rozas, Uday R. Savagaonkar, Simon P. Johnson, Vincent Scarlata, Michael A. Goldsmith, Ernie Brickell, Jiang Tao Li, Howard C. Herbert, Prashant Dewan, Stephen J. Tolopka, Gilbert Neiger, David Durham, Gary Graunke, Bernard Lint, Don A. Van Dyke, Joseph Cihula, Stalinselvaraj Jeyasingh, Stephen R. Van Doren, Dion Rodgers, John Garney, and Asher Altman. Method and apparatus to provide secure application execution, Dec 2009. US Patent 9,087,200.

Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 13:10, 2013.

MIT. Reference implementation of a Sanctum security monitor. `https://github.com/pwnall/sanctum`, 2017. [Online; accessed 1-Jan-2017].

Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox – practical cache attacks in JavaScript. *arXiv preprint arXiv:1502.07373*, 2015.

Peter Pessl, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Reverse engineering intel DRAM addressing and exploitation. *CoRR*, abs/1511.08756, 2015.

Stefan M. Petters and Georg Farber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Sixth International Conference on Real-Time Computing Systems and Applications*, pages 442–449. IEEE, 1999.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212. ACM, 2009.

Xiaoyu Ruan. *Platform Embedded Security Technology Revealed*. Apress, 2014. ISBN 978-1-4302-6571-9.

Joanna Rutkowska. Thoughts on Intel's upcoming software guard extensions (part 2). *Invisible Things Lab*, 2013.

Joanna Rutkowska and Rafał Wojtczuk. Preventing and detecting Xen hypervisor subversions. *Blackhat Briefings USA*, 2008.

Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling ways and associativity. In *Microarchitecture (MICRO), 2010 43rd annual IEEE/ACM International Symposium on*, pages 187–198. IEEE, 2010.

Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 57–68. ACM, 2011.

Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. `http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, Mar 2015. [Online; accessed 9-March-2015].

V. Shanbhogue, J. W. Brandt, and J. Wiedemeier. Protecting information processing system secrets from debug attacks, 2015. US Patent 8,955,144.

Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

G. Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, pages 9–14. ACM, 2007.

G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.

G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the* 32^{nd} *ISCA'05*. ACM, June 2005.

George Taylor, Peter Davies, and Michael Farmwald. The TLB slice - a low-cost high-speed address translation mechanism. *SIGARCH Computer Architecture News*, 18(2SI):355–363, 1990.

Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. *CoRR*, abs/1705.07289, 2017.

Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, 2007. ISBN 978-1-59593-706-3.

Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The RISC-V instruction set manual, volume I: User-level ISA, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html`.

Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanovic. The RISC-V instruction set manual volume II: Privileged architecture version 1.7. Technical Report UCB/EECS-2015-49, EECS Department, University of California, Berkeley, May 2015. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-49.html`.

Filip Wecherowski. A real SMM rootkit: Reversing and hooking BIOS SMI handlers. *Phrack Magazine*, 13(66), 2009.

Mark N. Wegman and J. Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981.

Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. *Invisible Things Lab*, 2009a.

Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel trusted execution technology. *Black Hat DC*, 2009b.

Rafal Wojtczuk and Joanna Rutkowska. Attacking intel txt via sinit code execution hijacking, 2011.

Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another way to circumvent Intel® trusted execution technology. *Invisible Things Lab*, 2009.

Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. IEEE – Institute of Electrical and Electronics Engineers, May 2015.

Yuval Yarom and Katrina E. Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013: 448, 2013.

Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

Marcelo Yuffe, Ernest Knoll, Moty Mehalel, Joseph Shor, and Tsvika Kurts. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264–266. IEEE, 2011.