

Foundations and Trends® in Electronic Design  
Automation  
Vol. 11, No. 1-2 (2017) 1–248  
© 2017 V. Costan, I. Lebedev, and S. Devadas  
DOI: 10.1561/10000000051



## **Secure Processors Part I: Background, Taxonomy for Secure Enclaves and Intel SGX 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

---

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                               | <b>2</b>  |
| 1.1      | Secure Remote Computation . . . . .               | 3         |
| 1.2      | SGX Lightning Tour . . . . .                      | 7         |
| 1.3      | Outline . . . . .                                 | 9         |
| <br>     |   |           |
| <b>2</b> | <b>A Primer on Computer System Architecture</b>   | <b>10</b> |
| 2.1      | Overview . . . . .                                | 11        |
| 2.2      | Computational Model . . . . .                     | 13        |
| 2.3      | Software Privilege Levels . . . . .               | 18        |
| 2.4      | Address Spaces . . . . .                          | 19        |
| 2.5      | Address Translation . . . . .                     | 22        |
| 2.6      | Execution Contexts . . . . .                      | 29        |
| 2.7      | Segment Registers . . . . .                       | 31        |
| 2.8      | Privilege Level Switching . . . . .               | 34        |
| 2.9      | An Overview of a Modern Computer System . . . . . | 38        |
| 2.10     | Out-of-Order and Speculative Execution . . . . .  | 44        |
| 2.11     | Memory Cache Subsystem . . . . .                  | 49        |
| 2.12     | Interrupts . . . . .                              | 62        |
| 2.13     | Platform Initialization (Booting) . . . . .       | 64        |
| 2.14     | CPU Microcode . . . . .                           | 69        |

|          |   |            |
|----------|---|------------|
| <b>3</b> | <b>A Primer on Security for Trusted Processors</b>                      | <b>79</b>  |
| 3.1      | Cryptographic Primitives . . . . .                                      | 80         |
| 3.2      | Cryptographic Constructs . . . . .                                      | 94         |
| 3.3      | Software Attestation Overview . . . . .                                 | 101        |
| 3.4      | Physical Attacks . . . . .  | 106        |
| 3.5      | Privileged Software Attacks . . . . .                                   | 111        |
| 3.6      | Software Attacks on Peripherals . . . . .                               | 112        |
| 3.7      | Address Translation Attacks . . . . .                                   | 117        |
| 3.8      | Cache Timing Attacks . . . . .  | 122        |
| <b>4</b> | <b>A Survey of Secure Processors</b>                                    | <b>128</b> |
| 4.1      | The IBM 4765 Secure Coprocessor . . . . .                               | 128        |
| 4.2      | ARM TrustZone . . . . .   | 132        |
| 4.3      | The XOM Architecture . . . . .  | 135        |
| 4.4      | The Trusted Platform Module (TPM) . . . . .                             | 136        |
| 4.5      | Intel's Trusted Execution Technology (TXT) . . . . .                    | 139        |
| 4.6      | The Aegis Secure Processor . . . . .                                    | 140        |
| 4.7      | The Bastion Architecture . . . . .                                      | 142        |
| 4.8      | Intel SGX . . . . .   | 143        |
| 4.9      | Sanctum . . . . .   | 144        |
| 4.10     | Ascend and Phantom . . . . .  | 145        |
| <b>5</b> | <b>The Software Isolation Container (As Exemplified by Intel's SGX)</b> | <b>147</b> |
| 5.1      | SGX Physical Memory Organization . . . . .                              | 149        |
| 5.2      | The Memory Layout of an SGX Enclave . . . . .                           | 153        |
| 5.3      | The Life Cycle of an SGX Enclave . . . . .                              | 161        |
| 5.4      | The Life Cycle of an SGX Thread . . . . .                               | 165        |
| 5.5      | EPC Page Eviction . . . . .   | 175        |
| 5.6      | SGX Enclave Measurement . . . . .                                       | 188        |
| 5.7      | SGX Enclave Versioning Support . . . . .                                | 195        |
| 5.8      | SGX Software Attestation . . . . .                                      | 208        |
| 5.9      | SGX Enclave Launch Control . . . . .                                    | 220        |
| <b>6</b> | <b>Conclusion</b>   | <b>230</b> |

|                        |            |
|------------------------|------------|
| <b>Acknowledgments</b> | <b>232</b> |
| <b>References</b>      | <b>233</b> |

## Abstract

This manuscript is the first 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. This manuscript first examines the relevant concepts in computer architecture and cryptography, and then surveys attack vectors and existing processor systems claiming security for remote computation and/or software isolation. This work examines in detail the modern isolation container (enclave) primitive as a means to minimize trusted software given practical trusted hardware and reasonable performance overhead. Specifically, this work examines in detail the programming model and software design considerations of Intel’s Software Guard Extensions (SGX), as it is an available and documented enclave-capable system.

Part II of this work is a deep dive into the implementation and security evaluation of two modern enclave-capable secure processor systems: SGX and MIT’s Sanctum. The complex but insufficient threat model employed by SGX motivates Sanctum, which achieves stronger security guarantees under software attacks with an equivalent programming model.

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

# 1

---

## Introduction

---

A user wishing to perform computation remotely faces a complex trade-off: how much trust can be placed in the remote system? How much of a performance overhead is considered acceptable for the given security properties? How strong an adversary can the remote system defend against? An ideal system would offer overhead-free trustworthy private remote computation with no assumptions of trust at all, yet no such system exists.

At one extreme, expensive cryptographic techniques including garbled circuits [Yao, 1986] and fully homomorphic encryption [Gentry, 2009] offer trust-free computation at prohibitive cost. A typical cloud computing scenario lies much closer to the opposite extreme: weak security guarantees achievable with minimal overhead assuming nearly unchecked trust in the remote system. This work aims to illustrate that significant security properties can be achieved given very modest trust in the remote system. A long lineage of secure processors explore the space of trusted hardware enabling inexpensive remote computation robust against a variety of threat models.

A rigorous conversation about security requires a precisely stated threat model: trusted hardware must be secure, meaning it must show

resilience against a well-specified threat model. For example, few systems can offer meaningful guarantees against an adversary capable of physically tampering with the system’s hardware. While the space of projects fitting the description of “secure processor” is large indeed, this work focuses on systems enabling *secure remote computation*, defined in § 1.1. Specifically, this work aims to illuminate the programming model, historical context, design decisions, and threat models relevant to secure software enclaves – the latest and so far the most capable paradigm for secure remote computation. We survey Intel’s Software Guard Extensions (SGX) and MIT’s Sanctum systems to exemplify enclave-capable systems.

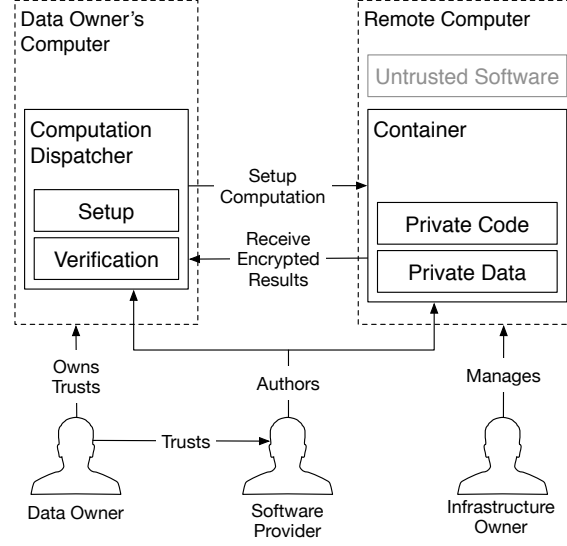
This work is presented in two parts, the first covering the technical background and taxonomy of computer architecture (§ 2) and security concepts (§ 3) as relevant to an in-depth discussion of secure processors. This same part presents a survey of prior work (§ 4) and an in-depth discussion of the programming model presented by secure software enclaves, as exemplified by Intel’s Software Guard Extensions (§ 5).

Part II [Costan et al., 2017] of this review is a deep dive into the implementation and security properties of two modern enclave-capable secure processor systems: SGX and MIT’s Sanctum. This work aims to rigorously analyze the security properties and trade-offs employed by the secure properties to achieve their stated goals.

## 1.1 Secure Remote Computation

Secure remote computation (Figure 1.1) is the problem of executing software on a remote computer **owned and maintained by an untrusted party**, with some integrity and confidentiality guarantees. In the general setting, secure remote computation is an unsolved problem. Fully Homomorphic Encryption [Gentry, 2009] addresses the problem for a limited family of computations, but has an impractical performance overhead [Naehrig et al., 2011].

Intel’s Software Guard Extensions (SGX) is the latest iteration in a long line of trusted computing (Figure 1.2) designs, which aim to solve the secure remote computation problem by leveraging trusted



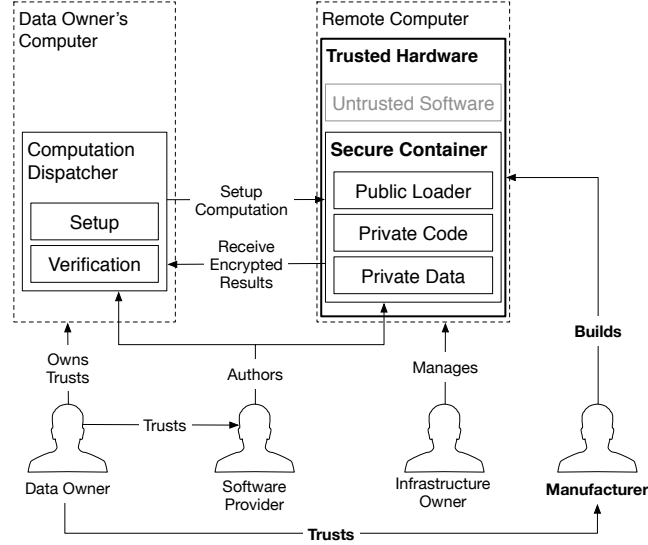
**Figure 1.1:** Secure remote computation. A user relies on a remote computer, owned by an untrusted party, to perform some computation on her data. The user has some assurance of the computation's integrity and confidentiality.

hardware in the remote computer. The trusted hardware establishes a secure container, and the remote computation service user uploads the desired computation and data into the secure container. The trusted hardware protects the confidentiality and integrity of data while the computation is being performed on it.

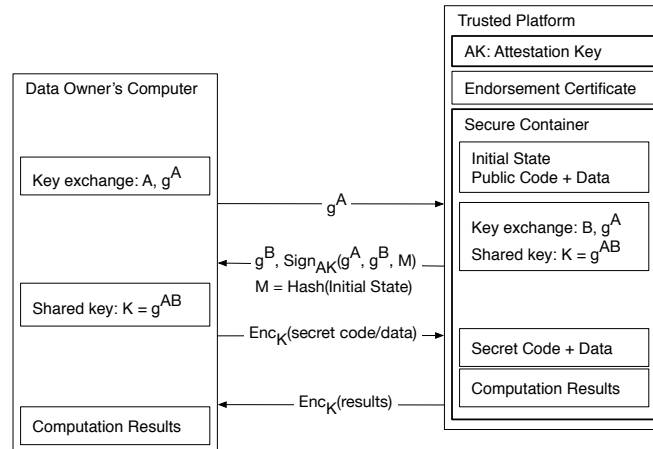
SGX, Sanctum, and similar work rely on *software attestation*, like their predecessors, the TPM [TCG, 2003] and TXT [Grawrock, 2009]. Attestation (Figure 1.3) proves to a user that she is communicating with a specific piece of software running in a secure container hosted by the trusted hardware. The proof is a cryptographic signature that certifies the hash of the secure container's contents. It follows that the remote computer's owner can load any software in a secure container, but the remote computation service user is able to refuse to send private data to a secure container with a hash that does not match an expected value.

The remote computation service user verifies the *attestation key* used to produce the signature against an *endorsement certificate* cre-





**Figure 1.2:** Trusted computing. The user trusts the manufacturer of a piece of hardware in the remote computer, and entrusts her data to a secure container hosted by the secure hardware.

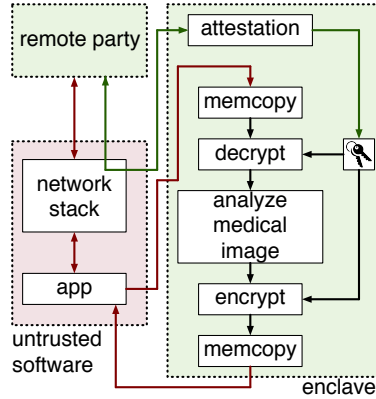


**Figure 1.3:** Software attestation proves to a remote computer that it is communicating with a specific secure container hosted by a trusted platform. The proof is an attestation signature produced by the platform's secret attestation key. The signature covers the container's initial state, a challenge nonce produced by the remote computer, and a message produced by the container.

ated by the trusted hardware’s manufacturer. The certificate states that the attestation key is only known to the trusted hardware, and only used for the purpose of attestation.

SGX stands out from its predecessors by the amount of code covered by the attestation, which is in the Trusted Computing Base (TCB) for the system using hardware protection. The attestations produced by the original TPM design covered the whole of the software running on a computer, and TXT attestations covered the code inside a VMX [Uhlig et al., 2005] virtual machine. In SGX, an *enclave* (secure container) only contains the private data in a computation, and the code that operates on it.

For example, a cloud service that performs image processing on confidential medical images could be implemented by having users upload encrypted images. The users would send the encryption keys to software running inside an enclave. The enclave would contain the code for decrypting images, the image processing algorithm, and the code for encrypting the results. The code that receives the uploaded encrypted images and stores them would be left outside the enclave. This example is illustrated in Figure 1.4.



**Figure 1.4:** An example software application that uses SGX to implement a private function analyzing a medical image.

An SGX-enabled processor protects the integrity and confidentiality of the computation inside an enclave by isolating the enclave’s code

and data from other software, including the operating system and hypervisor, and hardware devices attached to the system bus. At the same time, the SGX model remains compatible with the traditional software layering in the Intel architecture, where the OS kernel and hypervisor manage the computer's resources.

This work discusses the original version of SGX, also referred to as SGX 1. While SGX 2 brings very useful improvements for enclave authors, it is a small incremental improvement, from a design and implementation standpoint. After understanding the principles behind SGX 1 and its security properties, the reader should be well equipped to face Intel's reference documentation and learn about the changes brought by SGX 2 and more recent work.

## 1.2 SGX Lightning Tour

While this manuscript seeks to educate the reader of the challenges, history, and state of the art in secure processors for remote computation, this discussion is grounded in the example of Intel's Software Guard Extensions (SGX), as it is an available, documented, and modern system that aims to offer useful security guarantees to remotely executed programs. This section presents a brief overview of the SGX platform, directing the reader to other sections of the manuscript for a deeper look at each aspect of SGX.

SGX sets aside a memory region, called the *Processor Reserved Memory* (PRM, § 5.1). The CPU protects the PRM from all non-enclave memory accesses, including kernel, hypervisor and management engine (SMM, § 2.3) accesses, and DMA accesses (§ 2.9.1) from peripherals.

The PRM holds the *Enclave Page Cache* (EPC, § 5.1.1), which consists of 4 KB pages that store enclave code and data. The system software, which is untrusted, is in charge of assigning EPC pages to enclaves. The CPU tracks each EPC page's state in the *Enclave Page Cache Metadata* (EPCM, § 5.1.2), to ensure that each EPC page is assigned exclusively, belonging to exactly one enclave.

The initial code and data in an enclave is loaded by untrusted system software. During loading (§ 5.3), system software asks the CPU to copy data from unprotected memory (outside PRM) into EPC pages, and assigns the pages to the enclave being setup (§ 5.1.2). It follows that the initial enclave state is known to the system software.

After the enclave's pages are loaded into EPC, the system software asks the CPU to mark the enclave as initialized (§ 5.3), at which point application software may execute code inside the enclave. After an enclave is initialized, the loading mechanism briefly described above is no longer available to system software.

While an enclave is loaded, its contents and configuration are cryptographically hashed by the CPU. When the enclave is initialized, this hash is finalized, and becomes the enclave's *measurement hash* (§ 5.6).

A remote party can communicate with the enclave to perform *software attestation* (§ 5.8) to convince itself that it is communicating with an enclave that has a specific measurement hash, and is running in a secure environment.

Execution flow can only enter an enclave via special CPU instructions (§ 5.4), similar to the mode switching mechanism for transitioning between user and kernel modes of execution in a typical system. An enclave must execute in protected mode, at ring 3, and uses virtual address translation as set up by the OS kernel and hypervisor.

To avoid leaking private information, a CPU executing enclave code does not directly service any interrupt, fault (e.g., a page fault) or VM exit. Instead, the CPU first performs an Asynchronous Enclave Exit (§ 5.4.3) to switch from enclave code to ring 3 code, and then services the interrupt, fault, or VM exit given scrubbed fault information. The CPU performs an AEX by saving the CPU state into a predefined area inside the enclave and transferring control to a predefined address outside of the enclave, replacing CPU registers with synthetic values.

The allocation of EPC pages to enclaves is delegated to the OS kernel (or hypervisor). The OS communicates its allocation decisions to the SGX platform via special ring 0 CPU instructions (§ 5.3). The OS can also evict EPC pages into untrusted DRAM and later load them back, again using dedicated CPU instructions. SGX uses a cryptographic

mechanism to enforce the confidentiality, integrity and freshness of the evicted EPC pages while they are stored in untrusted memory.

### 1.3 Outline

Reasoning about the security properties of Intel’s SGX requires a significant amount of background information that is currently scattered across many sources. For this reason, a significant portion of this work is dedicated to summarizing this prerequisite knowledge.

§ 2 summarizes the relevant subset of modern computer architecture and the micro-architectural properties of recent Intel processors. § 3 outlines the landscape of trusted hardware systems, including cryptographic tools and relevant classes of attacks. Lastly, § 4 briefly describes other trusted hardware systems as context in which SGX was created.

Following this background information, § 5 provides a (sometimes painstakingly) detailed description of SGX’s programming model, largely drawing from Intel’s Software Development Manual.

A deep analysis of Intel’s enclave infrastructure is deferred to part II of this publication (§ II.2), and will analyze other public sources of information, such as Intel’s patents relevant to SGX, in order to fill in some of the missing detail in the SGX specification. This discussion is organized into an overview of Intel’s implementation of SGX (§ II.2.1), a discussion and analysis of the mechanism by which SGX offers memory access protection to an enclave (§ II.2.2, § II.2.3), and examines SGX as a system for remote attestation (§ II.2.5, § II.2.6). Finally, part II presents a security analysis of SGX overall, and discusses the classes of attacks against which SGX does not offer guarantees (§ II.2.7). The main focus of part II is a detailed review of SGX’s security properties to motivate and give context to the MIT Sanctum project (§ II.3) – a flexible, secure, and open source implementation of enclave-capable hardware that offers strong security guarantees against an insidious software adversary.

# 2

---

## A Primer on Computer System Architecture

---

Analyzing the security of a software system requires understanding the interactions between all parts of the software’s execution environment. This section attempts to summarize the architectural principles behind a modern processor, grounded in the specific example of the Intel Core architecture<sup>1</sup> (the widely accessible high-end computer system at the time of publication), which offers a complex tapestry of interacting sub-systems, subsets of which exemplify all architectural concepts required to reason about the security concepts in this survey.

In an effort to present an accessible view of the relevant aspects of computer architecture, this section presents each part of the computer system in introductory terms before refining these with the details of modern CPUs. Unless specified otherwise, the information here is summarized from Intel’s *Software Development Manual* (SDM) [Int, 2015g].

---

<sup>1</sup>In this paper, the term *Intel architecture* refers to the x86 architecture as described in Intel’s SDM. The entire x86 architecture is very complex, in part due to its native support for legacy software dating back to 1990. This work considers a subset - the microarchitecture as used by modern 64-bit software only.

## 2.1 Overview

A computer's main resources (§ 2.2) are *memory* and *processors*. On Intel computers, *Dynamic Random-Access Memory* (DRAM) modules (§ 2.9.1) provide the memory, and one or more CPU packages expose *logical processors* (§ 2.9.4). These resources are managed by *system software*. An Intel computer typically runs two kinds of system software, namely operating systems and hypervisors.

The Intel architecture was designed to support multiple concurrent application software instances, called *processes*. An *operating system* (§ 2.3), allocates the computer's resources to the running processes. Server computers, especially in cloud environments, may host multiple operating system instances concurrently. This is accomplished via a *hypervisor* (§ 2.3) scheduling the computer's resources among the operating system instances running on the computer.

System software uses virtualization techniques to isolate each piece of software that it manages (process or operating system) from the rest of the software running on the computer. This isolation is a key tool for keeping software complexity at manageable levels, as it allows application and OS developers to focus on their software, and ignore the interactions with other software that may run on the computer.

A key component of virtualization is address translation (§ 2.5), which is used to give software the impression that it owns all memory on the computer. Address translation provides isolation that prevents a piece of buggy or malicious software from directly damaging other software, by modifying its memory contents.

The other key component of virtualization is the set of software privilege levels (§ 2.3) enforced by the CPU. Hardware privilege separation ensures that a piece of buggy or malicious software cannot damage other software directly, or by interfering with the system software managing it.

Processes express their computing requirements by creating execution *threads*, which are assigned by the operating system to the computer's logical processors. A thread contains an execution context (§ 2.6), which is the information necessary to perform a com-

putation. For example, an execution context stores the address of the next instruction that will be executed by the processor.

Operating systems give each process the illusion that it has an unbounded quantity of logical processors at its disposal, and multiplex the physically available logical processors between the threads created by each process. Modern operating systems implement *preemptive multithreading*, where the logical processors are rotated between all threads on a system every few milliseconds. Changing the thread assigned to a logical processor is accomplished by a context switch (§ 2.6).

Hypervisors expose a fixed number of virtual processors (vCPUs) to each operating system, and also use context switching to schedule the physical cores of a computer among the vCPUs presented to the guest operating systems.

The execution core in a logical processor can execute instructions and consume data at a much faster rate than DRAM can supply them. Many of the complexities in modern computer architectures stem from architectural mechanisms to close this gap. Recent Intel CPUs rely on hyper-threading (§ 2.9.4), out-of-order execution (§ 2.10), and caching (§ 2.11) to efficiently utilize available memory bandwidth, all of which have security implications.

An Intel processor contains many hierarchical levels of intermediate memory that are much faster than DRAM, but are also orders of magnitude smaller. The fastest of these is the logical processor's register file (§ 2.2, § 2.4, § 2.6). Other intermediate memory structures are various caches (§ 2.11). The Intel architecture requires application software to explicitly manage the register file, which serves as a high-speed scratch space, while caches transparently reduce expected latency of a given DRAM request, and are largely invisible to software.

Intel computers have multiple logical processors. As a consequence, they also have multiple caches distributed across the processor die. On multi-socket systems, the caches are distributed across multiple CPU packages. Therefore, Intel systems use a cache coherence mechanism (§ 2.11.3) to ensure that all caches have the same view of DRAM, allowing programmers to build software that is unaware of caching. However, cache coherence does not cover function-specific caches used



by address translation (TLBs, § 2.11.5), and system software must take special measures to keep these caches consistent.

CPUs communicate with the outside world via I/O devices (also known as peripherals), such as network interface cards and display adapters (§ 2.9). Conceptually, the CPU communicates with the DRAM modules and the I/O devices via a *system bus* that connects all these components.

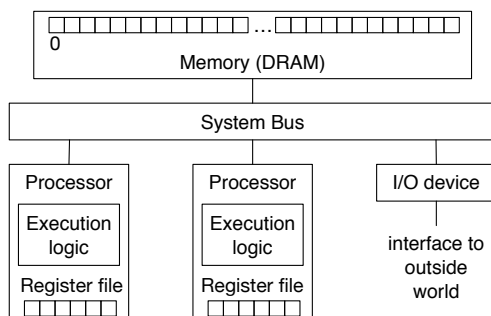
Software written for the Intel architecture communicates with I/O devices via the I/O address space (§ 2.4) and via the memory address space, which is primarily used to access DRAM. System software must configure the CPU's caches (§ 2.11.4) to recognize the memory address ranges used by I/O devices. Devices can notify the CPU of the occurrence of events by dispatching interrupts (§ 2.12), which cause a logical processor to stop executing its current thread, and invoke a special handler in the system software (§ 2.8.2).

Intel systems have a highly complex computer initialization sequence (§ 2.13), due to the need to support a large variety of peripherals, as well as a multitude of operating systems targeting different versions of the architecture. This initialization sequence poses numerous challenges to building a secure system around an Intel CPU, and has facilitated many security compromises (§ 2.3).

Intel's engineers use the processor's microcode facility (§ 2.14) to implement the more complicated aspects of the Intel architecture, which greatly helps manage hardware complexity. The microcode is completely transparent to software developers, and its design is largely undocumented. However, in order to reason about the feasibility of any proposals to alter the Intel platform, one must be aware of the limits of microcode, and understand the space of changes that can be implemented without modifying the underlying hardware.

## 2.2 Computational Model

A simplified model presented in Figure 2.1 frames this work. Following sections refine this model into a detailed description of the Intel architecture.



**Figure 2.1:** A high-level view of the architected memory resources of a CPU. The system bus also links memory-mapped and I/O devices, such as keyboards, which are also connected to the processor via the system bus.

The building blocks for the model presented here come from the book [Saltzer and Kaashoek, 2009], which introduces the key abstractions in a computer system, and then focuses on the techniques used to build software systems on top of these abstractions.

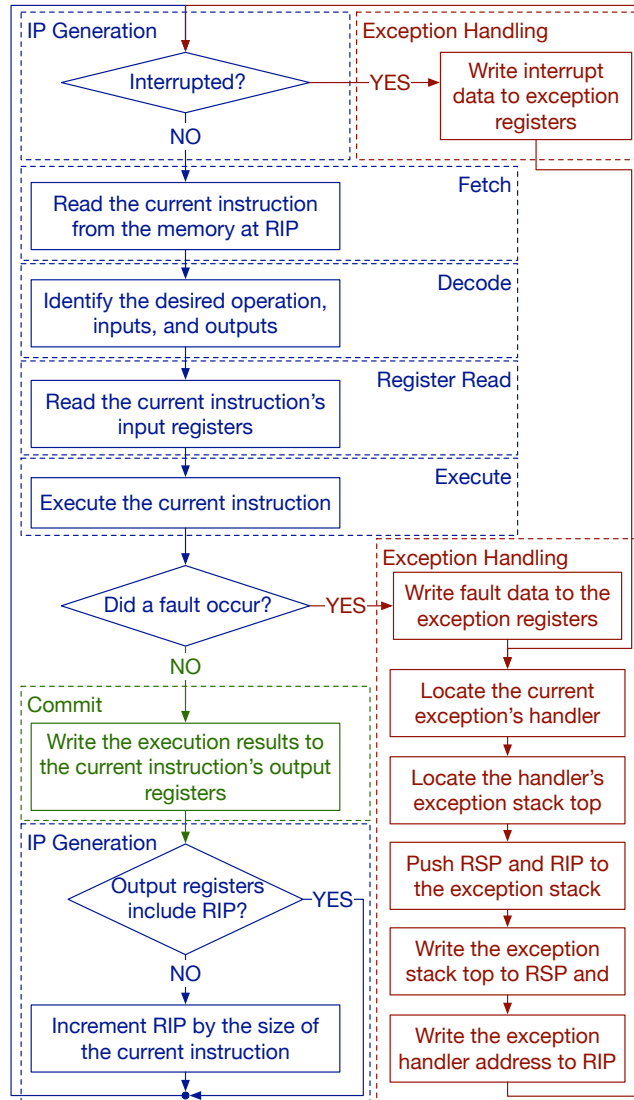
The memory is an array of individually addressable storage locations, indexed via natural numbers, and implements the abstraction depicted in Figure 2.2. Its salient feature is that the result of reading a memory cell at an address must equal the most recent value written to that memory cell by a given program.

|   |
|---|
| $\text{WRITE}(\text{addr}, \text{value}) \rightarrow \emptyset$<br>Store <i>value</i> in the storage cell identified by <i>addr</i> .           |
| $\text{READ}(\text{addr}) \rightarrow \text{value}$<br>Return the <i>value</i> argument to the most recent WRITE call referencing <i>addr</i> . |

**Figure 2.2:** The memory abstraction

A logical processor repeatedly reads *instructions* from the computer’s memory and executes them, according to the flowchart in Figure 2.3.

The processor has an internal memory, referred to as the *register file*. The register file consists of *Static Random Access Memory* (SRAM)



**Figure 2.3:** A processor fetches instructions from the memory and executes them. The RIP register holds the address of the instruction to be executed.

cells, generally known as *registers*, which are significantly faster than DRAM cells, but also a lot more expensive.

An instruction performs a simple computation on its inputs and stores the result in an output location. The processor's registers make up an *execution context* that provides the inputs and stores the outputs for most instructions. For example, `ADD RDX, RAX, RBX` performs an integer addition, where the inputs are the registers RAX and RBX, and the result is stored in the output register RDX.

The registers mentioned in Figure 2.3 are the *instruction pointer* (RIP), which stores the memory address of the next instruction to be executed by the processor, and the *stack pointer* (RSP), which stores the memory address of the topmost element in the call stack used by the processor's procedural programming support. The other execution context registers are described in § 2.4 and § 2.6.

Under normal circumstances, the processor repeatedly reads an instruction from the memory address stored in RIP, executes the instruction, and updates RIP to point to the following instruction. Unlike many RISC architectures, the Intel architecture uses a variable-size instruction encoding, so the size of an instruction is not known until the instruction has been read from memory.

While executing an instruction, the processor may encounter a *fault*, which is a situation where the instruction's preconditions are not met. When a fault occurs, the instruction does not store a result in the output location. Instead, the instruction's result is considered to be the fault that occurred. For example, an integer division instruction `DIV` where the divisor is zero results in a Division Fault (`#DIV`).

When an instruction results in a fault, the processor stops its normal execution flow, and performs the fault handler process documented in § 2.8.2. In a nutshell, the processor first looks up the address of the code that will handle the fault, based on the fault's nature, and sets up the execution environment in preparation to execute the fault handler.

The processors are connected to each other and to the memory via a *system bus*, which is a broadcast network that implements the abstraction in Figure 2.4.

|  |
|--|
| SEND( $op, addr, data$ ) $\rightarrow \emptyset$<br>Place a message containing the operation code $op$ , the bus address $addr$ , and the value $data$ on the bus. |
| READ() $\rightarrow (op, addr, value)$<br>Return the message that was written on the bus at the beginning of this clock cycle.                                     |

**Figure 2.4:** The system bus abstraction.

During each clock cycle, at most one of the devices connected to the system bus can send a message, which is received by all other devices connected to the bus. Each device attached to the bus decodes the operation codes and addresses of all messages sent on the bus and ignores the messages that do not require its involvement.

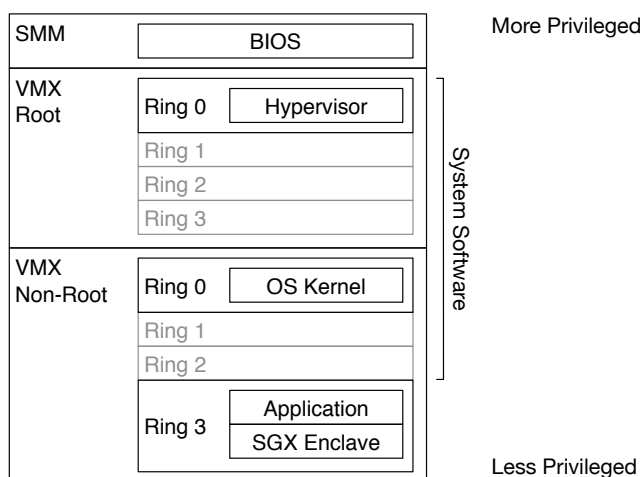
For example, when the processor wishes to read a memory location, it sends a message with the operation code READ-REQUEST and the bus address corresponding to the desired memory location. The memory sees the message on the bus and performs the READ operation. At a later time, the memory responds by sending a message with the operation code READ-RESPONSE, the same address as the request, and the data value set to the result of the READ operation.

The computer communicates with the outside world via I/O devices, such as keyboards, displays, and network cards, which are connected to the system bus. Devices mostly respond to requests issued by the processor. However, devices also have the ability to issue *interrupt requests* that notify the processor of outside events, such as the user pressing a key on a keyboard.

Interrupt triggering is discussed in § 2.12. On modern systems, devices send interrupt requests by issuing writes to special bus addresses. Interrupts are considered to be *hardware exceptions*, just like faults, and are handled in a similar manner.

## 2.3 Software Privilege Levels

In an Infrastructure-as-a-Service (IaaS) cloud environment, such as Amazon EC2, commodity CPUs run software at four different privilege levels, shown in Figure 2.5.



**Figure 2.5:** The privilege levels in the x86 architecture, and the software that typically runs at each security level.

Each privilege level is strictly more powerful than the ones below it, so a piece of software can freely read and modify the code and data running at less privileged levels. Therefore, a software module can be compromised by any piece of software running at a higher privilege level. It follows that a software module implicitly trusts all software running at more privileged levels, and a system's security analysis must take into account the software at all privilege levels.

*System Management Mode* (SMM) is intended for use by the motherboard manufacturers to implement features such as fan control and deep sleep, and/or to emulate missing hardware. Therefore, the bootstrapping software (§ 2.13) in the computer's firmware is responsible for setting up a continuous subset of DRAM as *System Management RAM* (SMRAM), and for loading all of the code that needs to run in SMM mode into SMRAM. The SMRAM enjoys special hardware protections that prevent less privileged software from accessing the SMM

code. IaaS cloud providers allow their customers to run their operating system of choice in a virtualized environment. Hardware virtualization [Uhlig et al., 2005], called *Virtual Machine Extensions* (VMX) by Intel, adds support for a *hypervisor*, also called a *Virtual Machine Monitor* (VMM) in the Intel documentation. The hypervisor runs at a higher privilege level (VMX root mode) than the operating system, and is responsible for allocating hardware resources across multiple operating systems that share the same physical machine. The hypervisor uses the CPU’s hardware virtualization features to make each operating system believe it is running in its own computer, called a *virtual machine* (VM). Hypervisor code generally runs at ring 0 in VMX root mode. Hypervisors that run in VMX root mode and take advantage of hardware virtualization generally have better performance and a smaller codebase than hypervisors based on binary translation [Rosenblum and Garfinkel, 2005]. The systems research literature recommends breaking up an operating system into a small *kernel*, which runs at a high privilege level, known as the *kernel mode* or *supervisor mode* and, in the Intel architecture, as *ring 0*. The kernel allocates the computer’s resources to the other system components, such as device drivers and services, which run at lower privilege levels. However, for performance reasons<sup>2</sup>, mainstream operating systems have large amounts of code running at ring 0. Their *monolithic kernels* include device drivers, filesystem code, networking stacks, and video rendering functionality. Application code, such as a Web server or a game client, runs at the lowest privilege level, referred to as *user mode* (*ring 3* in the Intel architecture). In IaaS cloud environments, the virtual machine images provided by customers run in VMX non-root mode, so the kernel runs in VMX non-root ring 0, and the application code runs in VMX non-root ring 3.

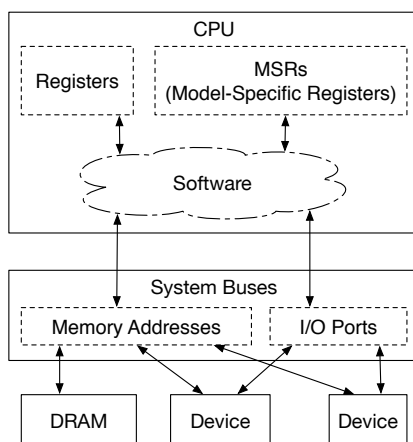
## 2.4 Address Spaces

Software written for the Intel architecture accesses the computer’s resources using four distinct physical address spaces, shown in Figure 2.6.

---

<sup>2</sup>Calling a procedure in a different ring is much slower than calling code at the same privilege level.

The address spaces overlap partially, in both purpose and contents, which can lead to confusion. This section gives a high-level overview of the physical address spaces defined by the Intel architecture, with an emphasis on their purpose and the methods used to manage them.



**Figure 2.6:** The four physical address spaces used by an Intel CPU. The registers and MSRs are internal to the CPU, while the memory and I/O address spaces are used to communicate with DRAM and other devices via system buses.

The *register* space consists of names that are used to access the CPU's register file, which is the only memory that operates at the CPU's clock frequency and can be used without any latency penalty. The register space is defined by the CPU's architecture, and documented in the SDM.

Some registers, such as the *Control Registers* (CRs) play specific roles in configuring the CPU's operation. For example, CR3 plays a central role in address translation (§ 2.5). These registers can only be accessed by system software. The rest of the registers make up an application's *execution context* (§ 2.6), which is essentially a high-speed scratch space. These registers can be accessed at all privilege levels, and their allocation is managed by the software's compiler. Many CPU instructions only operate on data in registers, and only place their results in registers.



The *memory* space, generally referred to as *the address space*, or *the physical address space*, consists of  $2^{36}$  (64 GB) -  $2^{40}$  (1 TB) addresses. The memory space is primarily used to access DRAM, but it is also used to communicate with *memory-mapped devices* that read memory requests off a system bus and write replies for the CPU. Some CPU instructions can read their inputs from the memory space, or store the results using the memory space.

A better-known example of memory mapping is that at computer startup, memory addresses 0xFFFFF000 - 0xFFFFFFFF (the 64 KB of memory right below the 4 GB mark) are mapped to a flash memory device that holds the first stage of the code that bootstraps the computer.

The memory space is partitioned between devices and DRAM by the computer's firmware during the bootstrapping process. Sometimes, system software includes motherboard-specific code that modifies the memory space partitioning. The OS kernel relies on address translation, described in § 2.5, to control the applications' access to the memory space. The hypervisor relies on the same mechanism to control the guest OSs.

The *input/output* (I/O) space consists of  $2^{16}$  I/O addresses, usually called *ports*. The I/O ports are used exclusively to communicate with devices. The CPU provides specific instructions for reading from and writing to the I/O space. I/O ports are allocated to devices by formal or de-facto standards. For example, ports 0xCF8 and 0xCFC are always used to access the PCI express (§ 2.9.1) configuration space.

The CPU implements a mechanism for system software to provide fine-grained I/O access to applications. However, all modern kernels restrict application software from accessing the I/O space directly, in order to limit the damage potential of application bugs.

The *Model-Specific Register* (MSR) space consists of  $2^{32}$  MSRs, which are used to configure the CPU's operation. The MSR space was initially intended for the use of CPU model-specific firmware, but some MSRs have been promoted to *architectural MSR* status, making their semantics a part of the Intel architecture. For example, architectural MSR 0x10 holds a high-resolution monotonically increasing time-stamp counter.

The CPU provides instructions for reading from and writing to the MSR space. The instructions can only be used by system software. Some MSRs are also exposed by instructions accessible to applications. For example, applications can read the time-stamp counter via the RDTSC and RDTSCP instructions, which are very useful for benchmarking and optimizing software.

## 2.5 Address Translation

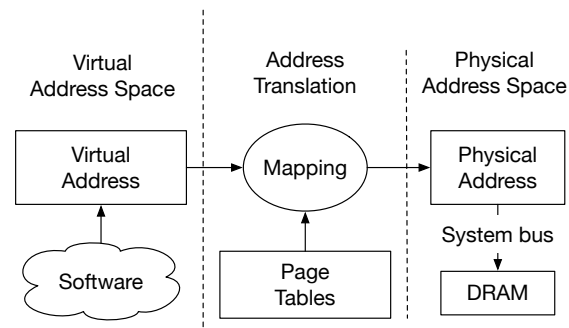
System software relies on the CPU's address translation mechanism for implementing isolation among less privileged pieces of software (applications or operating systems). Virtually all secure architecture designs bring changes to address translation. We summarize the Intel architecture's address translation features that are most relevant when establishing a system's security properties, and refer the reader to [Jacob and Mudge, 1998] for a more general presentation of address translation concepts and its other uses.

### 2.5.1 Address Translation Concepts

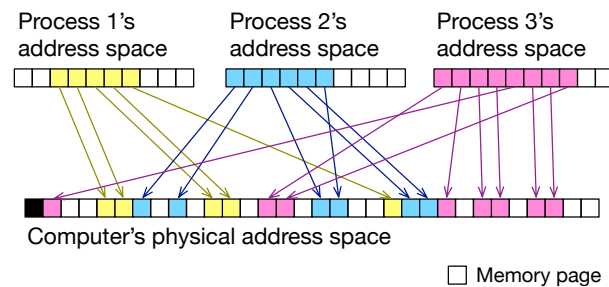
From a systems perspective, address translation is a layer of indirection (shown in Figure 2.7) between the *virtual addresses*, which are used by a program's memory load and store instructions, and the *physical addresses*, which reference the physical address space (§ 2.4). The mapping between virtual and physical addresses is defined by *page tables*, which are managed by the system software.

Operating systems use address translation to implement the *virtual memory abstraction*, illustrated by Figure 2.8. The virtual memory abstraction exposes the same interface as the memory abstraction in § 2.2, but each process uses a separate virtual address space that only references the memory allocated to that process. From an application developer standpoint, virtual memory can be modeled by pretending that each process runs on a separate computer and has its own DRAM.

Address translation is used by the operating system to multiplex DRAM among multiple application processes, isolate the processes from each other, and prevent application code from accessing memory-



**Figure 2.7:** Virtual addresses used by software are translated into physical memory addresses using a mapping defined by the page tables.



**Figure 2.8:** The virtual memory abstraction gives each process its own virtual address space. The operating system multiplexes the computer's DRAM between the processes, while application developers build software as if it owns the entire computer's memory.

mapped devices directly. The latter two protection measures prevent an application's bugs from impacting other applications or the OS kernel itself. Hypervisors also use address translation, to divide the DRAM among operating systems that run concurrently, and to virtualize memory-mapped devices.

The address translation mode used by 64-bit operating systems, called IA-32e by Intel's documentation, maps 48-bit *virtual addresses* to *physical addresses* of at most 52 bits<sup>3</sup>. The translation process, illustrated in Figure 2.9, is carried out by dedicated hardware in the CPU, which is referred to as the *address translation unit* or the *memory management unit* (MMU).

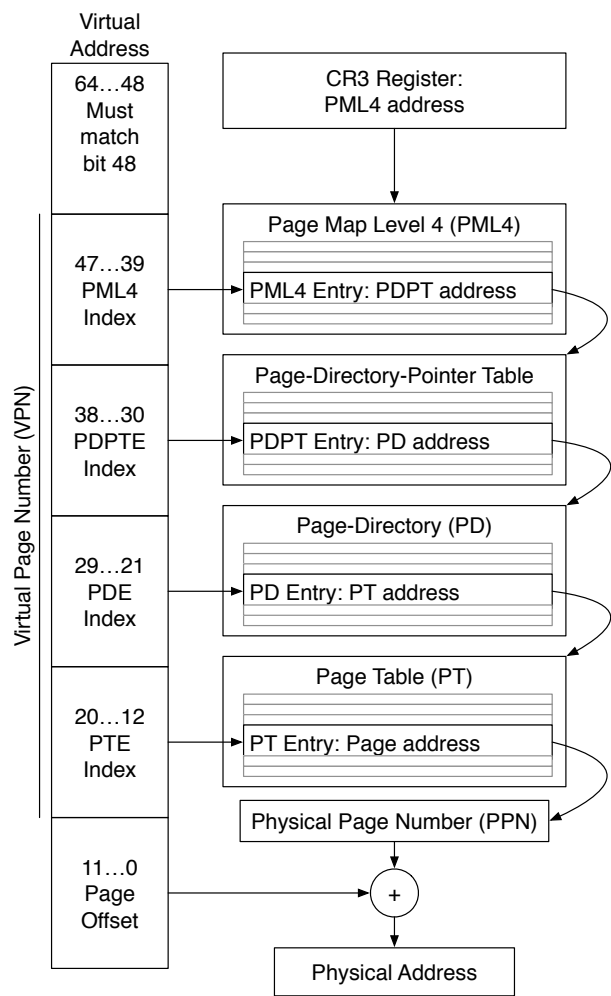
The bottom 12 bits of a virtual address are not changed by the translation. The top 36 bits are grouped into four 9-bit indexes, which are used to index into the page tables. Despite its name, the page tables data structure closely resembles a full 512-ary search tree where nodes have fixed keys. Each node is represented in DRAM as an array of 512 8-byte entries that contain the physical addresses of the next-level children as well as some flags. The physical address of the root node is stored in the CR3 register. The arrays in the last-level nodes contain the physical addresses that are the result of the address translation.

The address translation function, which does not change the bottom bits of addresses, partitions the memory address space into *pages*. A page is the set of all memory locations that only differ in the bottom bits which are not impacted by address translation, so all memory addresses in a virtual page translate to corresponding addresses in the same physical page. From this perspective, the address translation function can be seen as a mapping between *Virtual Page Numbers* (VPN) and *Physical Page Numbers* (PPN), as shown in Figure 2.10.

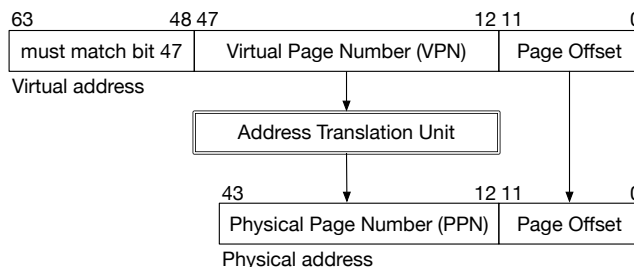
In addition to isolating application processes, operating systems also use the address translation feature to run applications whose collective memory demands exceed the amount of DRAM installed in the computer. The OS evicts infrequently used memory pages from DRAM to a larger (but slower) memory, such as a hard disk drive

---

<sup>3</sup>The size of a physical address is CPU-dependent, and is 40 bits for recent desktop CPUs and 44 bits for recent high-end server CPUs.



**Figure 2.9:** IA-32e address translation takes in a 48-bit virtual address and outputs a 52-bit physical address.



**Figure 2.10:** Address translation can be seen as a mapping between virtual page numbers and physical page numbers.

(HDD) or solid-state drive (SSD). For historical reason, this slower memory is referred to as the *disk*.

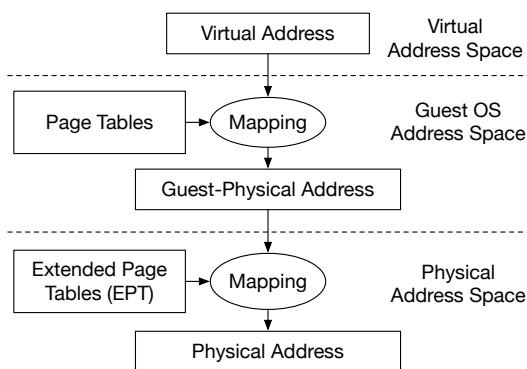
The operating system’s ability to over-commit DRAM is often called *page swapping*, for the following reason. When an application process attempts to access a page that has been evicted, the OS “steps in” and reads the missing page back into DRAM. In order to do this, the OS may need to evict a different page from DRAM, effectively swapping the contents of a DRAM page with a page from disk. The details behind this high-level description are covered in the following sections.

The CPU’s address translation is also referred to as “paging”, which is a shorthand for “page swapping”.

## 2.5.2 Address Translation and Virtualization

Computers that take advantage of hardware virtualization use a hypervisor to host multiple operating systems simultaneously. This creates some tension, because each operating system was written under the assumption that it owns the entire computer’s DRAM. The tension is solved by a second layer of address translation, illustrated in Figure 2.11.

When a hypervisor is active, the page tables set up by an operating system map between virtual addresses and *guest-physical addresses* in a *guest-physical address space*. The hypervisor multiplexes the computer’s DRAM between the operating systems’ guest-physical address spaces via the second layer of address translations, which



**Figure 2.11:** Virtual addresses used by software are translated into physical memory addresses using a mapping defined by the page tables.

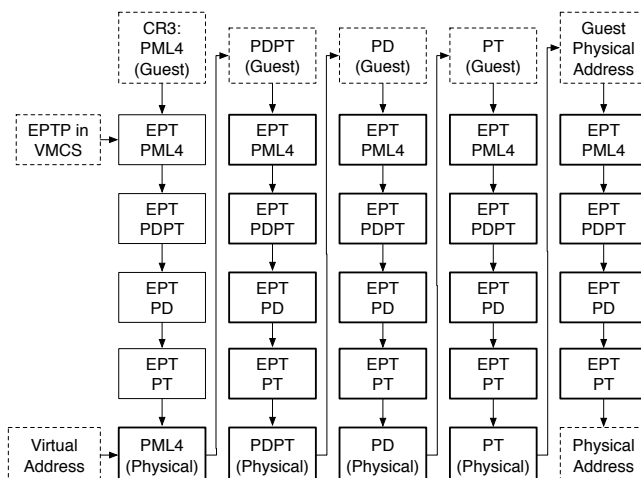
uses *extended page tables* (EPT) to map guest-physical addresses to physical addresses.

The EPT uses the same data structure as the page tables, so the process of translating guest-physical addresses to physical addresses follows the same steps as IA-32e address translation. The main difference is that the physical address of the data structure's root node is stored in the extended page table pointer (EPTP) field in the *Virtual Machine Control Structure* (VMCS) for the guest OS. Figure 2.12 illustrates the address translation process in the presence of hardware virtualization.

### 2.5.3 Page Table Attributes

Each page table entry contains a physical address, as shown in Figure 2.9, and some Boolean values that are referred to as *flags* or *attributes*. The following attributes are used to implement page swapping and software isolation.

The *present* (P) flag is set to 0 to indicate unused parts of the address space, which do not have physical memory associated with them. The system software also sets the P flag to 0 for pages that are evicted from DRAM. When the address translation unit encounters a zero P flag, it aborts the translation process and issues a hardware exception, as described in § 2.8.2. This hardware exception gives system software an opportunity to step in and bring an evicted page back into DRAM.



**Figure 2.12:** Address translation when hardware virtualization is enabled. The kernel-managed page tables contain guest-physical addresses, so each level in the kernel's page table requires a full walk of the hypervisor's extended page table (EPT). A translation requires up to 20 memory accesses (the bold boxes), assuming the physical address of the kernel's PML4 is cached.

The *accessed* (A) flag is set to 1 by the CPU whenever the address translation machinery reads a page table entry, and the *dirty* (D) flag is set to 1 by the CPU when an entry is accessed by a memory write operation. The A and D flags give the hypervisor and kernel insight into application memory access patterns and inform the algorithms that select the pages that get evicted from RAM.

The main attributes supporting software isolation are the *writable* (W) flag, which can be set to 0 to prohibit<sup>4</sup> writes to any memory location inside a page, the *disable execution* (XD) flag, which can be set to 1 to prevent instruction fetches from a page, and the *supervisor* (S) flag, which can be set to 1 to prohibit any accesses from application software running at ring 3.

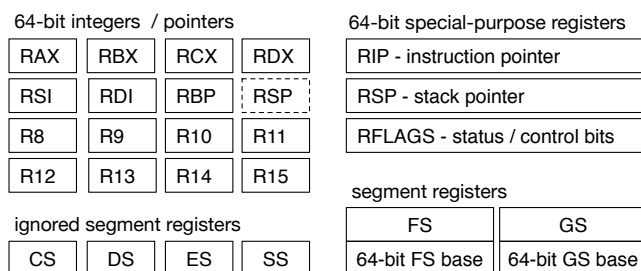
<sup>4</sup>Writes to non-writable pages result in #GP exceptions (§ 2.8.2).



## 2.6 Execution Contexts

Application software targeting the 64-bit Intel architecture uses a variety of CPU registers to interact with the processor’s features, shown in Figure 2.13 and Table 2.1. The values in these registers make up an application thread’s state, or *execution context*.

OS kernels multiplex each logical processor (§ 2.9.4) between multiple software threads by *context switching*, namely saving the values of the registers that make up a thread’s execution context, and replacing them with another thread’s previously saved context. Context switching also plays a part in executing code inside secure containers, so its design has security implications.



**Figure 2.13:** CPU registers in the 64-bit Intel architecture. RSP can be used as a general-purpose register (GPR), e.g., in pointer arithmetic, but it always points to the top of the program’s stack. Segment registers are covered in § 2.7.

Integers and memory addresses are stored in 16 *general-purpose registers* (GPRs). The first 8 GPRs have historical names: RAX, RBX, RCX, RDX, RSI, RDI, RSP, and RBP, because they are extended versions of the 32-bit Intel architecture’s GPRs. The other 8 GPRs are simply known as R9-R16. RSP is designated for pointing to the top of the procedure call stack, which is simply referred to as *the stack*. RSP and the stack that it refers to are automatically read and modified by the CPU instructions that implement procedure calls, such as **CALL** and **RET** (return), and by specialized stack handling instructions such as **PUSH** and **POP**.

All applications also use the RIP register, which contains the address of the currently executing instruction, and the RFLAGS register,

whose bits (e.g., the carry flag - CF) are individually used to store comparison results and control various instructions.

Software may use other registers to interact with specific processor features, some of which are shown in Table 2.1.

**Table 2.1:** Sample feature-specific Intel architecture registers.

| Feature | Registers                      | XCR0 bit |
|---------|--------------------------------|----------|
| FPU     | FP0 - FP7, FSW, FTW            | 0        |
| SSE     | MM0 - MM7, XMM0 - XMM15, XMCSR | 1        |
| AVX     | YMM0 - YMM15                   | 2        |
| MPX     | BND0 - BND 3                   | 3        |
| MPX     | BNDCFGU, BNDSTATUS             | 4        |
| AVX-512 | K0 - K7                        | 5        |
| AVX-512 | ZMM0_H - ZMM15_H               | 6        |
| AVX-512 | ZMM16 - ZMM31                  | 7        |
| PK      | PKRU                           | 9        |

The Intel architecture provides a future-proof method for an OS kernel to save the values of feature-specific registers used by an application. The **XSAVE** instruction takes in a *requested-feature bitmap* (RFBM), and writes the registers used by the features whose RFBM bits are set to 1 in a memory area. The memory area written by **XSAVE** can later be used by the **XRSTOR** instruction to load the saved values back into feature-specific registers. The memory area includes the RFBM given to **XSAVE**, so **XRSTOR** does not require an RFBM input.

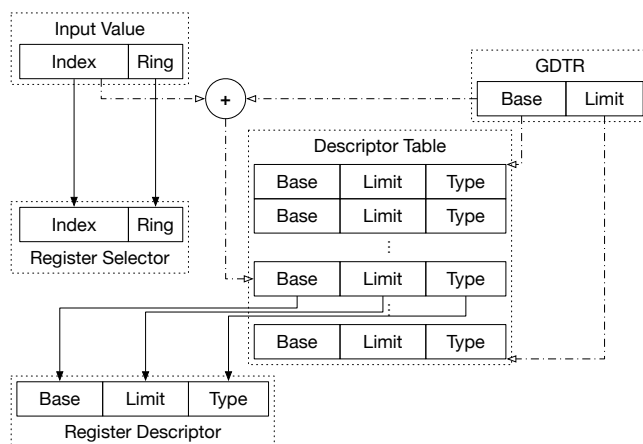
Application software declares the features that it plans to use to the kernel, so the kernel knows what **XSAVE** bitmap to use when context-switching. When receiving the system call, the kernel sets the XCR0 register to the feature bitmap declared by the application. The CPU generates a fault if application software attempts to use features that are not enabled by XCR0, so applications cannot modify feature-specific registers that the kernel wouldn't take into account when context-switching. The kernel can use the **CPUID** instruction to learn the size of the **XSAVE** memory area for a given feature bitmap,

and compute how much memory it needs to allocate for the context of each of the application's threads.

## 2.7 Segment Registers

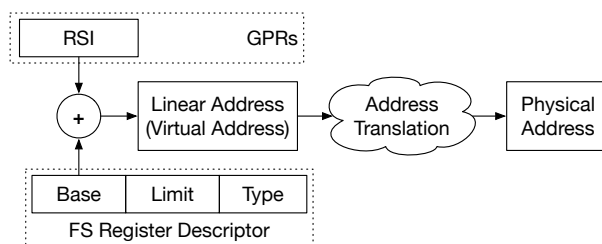
The Intel 64-bit architecture gained widespread adoption thanks to its ability to run software targeting the older 32-bit architecture side-by-side with 64-bit software [Shankland, 2005]. This ability comes at the cost of some warts. While most of these warts can be ignored while reasoning about the security of 64-bit software, the segment registers and vestigial segmentation model must be understood. The semantics of the Intel architecture's instructions include the implicit use of a few segments which are loaded into the processor's *segment registers* shown in Figure 2.13. Code fetches use the *code segment* (CS). Instructions that reference the stack implicitly use the *stack segment* (SS). Memory references implicitly use the *data segment* (DS) or the *destination segment* (ES). Via segment override prefixes, instructions can be modified to use the unnamed segments FS and GS for memory references. Modern operating systems effectively disable segmentation by covering the entire addressable space with one segment, which is loaded in CS, and one data segment, which is loaded in SS, DS and ES. The FS and GS registers store segments covering *thread-local storage* (TLS). Due to the Intel architecture's 16-bit origins, segment registers are exposed as 16-bit values, called *segment selectors*. The top 13 bits in a selector are an index in a *descriptor table*, and the bottom 2 bits are the selector's ring number, which is also called requested privilege level (RPL) in the Intel documentation. Also, modern system software only uses rings 0 and 3 (see § 2.3). Each segment register has a hidden *segment descriptor*, which consists of a *base address*, *limit*, and type information, such as whether the descriptor should be used for executable code or data. Figure 2.14 shows the effect of loading a 16-bit selector into a segment register. The selector's index is used to read a descriptor from the descriptor table and copy it into the segment register's hidden descriptor.

In 64-bit mode, all segment limits are ignored. The base addresses in most segment registers (CS, DS, ES, SS) are ignored. The base ad-



**Figure 2.14:** Loading a segment register. The 16-bit value loaded by software is a selector consisting of an index and a ring number. The index selects a GDT entry, which is loaded into the descriptor part of the segment register.

addresses in FS and GS are used, in order to support thread-local storage. Figure 2.15 outlines the address computation in this case. The instruction's address, named *logical address* in the Intel documentation, is added to the base address in the segment register's descriptor, yielding the virtual address, also named *linear address*. The virtual address is then translated (§ 2.5) to a physical address.



**Figure 2.15:** Example address computation process for `MOV FS:[RDX], 0`. The segment's base address is added to the address in RDX before address translation (§ 2.5) takes place.

Outside the special case of using FS or GS to reference thread-local storage, the logical and virtual (linear) addresses match. Therefore, most of the time, we can get away with completely ignoring segmen-

tation. In these cases, we use the term “virtual address” to refer to both the virtual and the linear address. Even though CS is not used for segmentation, 64-bit system software needs to load a valid selector into it. The CPU uses the ring number in the CS selector to track the current privilege level, and uses one of the type bits to know whether it’s running 64-bit code, or 32-bit code in compatibility mode.

The DS and ES segment registers are completely ignored, and can have null selectors loaded in them. The CPU loads a null selector in SS when switching privilege levels, discussed in § 2.8.2.

Modern kernels only use one descriptor table, the *Global Descriptor Table* (GDT), whose virtual address is stored in the GDTR register. Table 2.2 shows a typical GDT layout that can be used by 64-bit kernels to run both 32-bit and 64-bit applications.

**Table 2.2:** A typical GDT layout in the 64-bit Intel Architecture.

| Descriptor            | Selector               |
|-----------------------|------------------------|
| Null (must be unused) | 0                      |
| Kernel code           | 0x08 (index 1, ring 0) |
| Kernel data           | 0x10 (index 2, ring 0) |
| User code             | 0x1B (index 3, ring 3) |
| User data             | 0x1F (index 4, ring 3) |
| TSS                   | 0x20 (index 5, ring 0) |

The last entry in Table 2.2 is a descriptor for the *Task State Segment* (TSS), which was designed to implement hardware context switching, named *task switching* in the Intel documentation. The descriptor is stored in the *Task Register* (TR), which behaves like the other segment registers described above. Task switching was removed from the 64-bit architecture, but the TR segment register was preserved, and it points to a repurposed TSS data structure. The 64-bit TSS contains an *I/O map*, which indicates what parts of the I/O address space can be accessed directly from ring 3, and the *Interrupt Stack Table* (IST), which is used for privilege level switching (§ 2.8.2). Modern operating systems do not allow application software any direct access to the I/O address space, so the kernel sets up a single TSS that is

loaded into TR during early initialization, and used to represent all applications running under the OS.

## **2.8 Privilege Level Switching**

Any architecture that implements software privilege levels must provide a method for less privileged software to invoke the services of software with higher privilege. For example, application software needs the OS kernel's assistance to perform network or disk I/O, as that requires access to privileged memory or to the I/O address space.

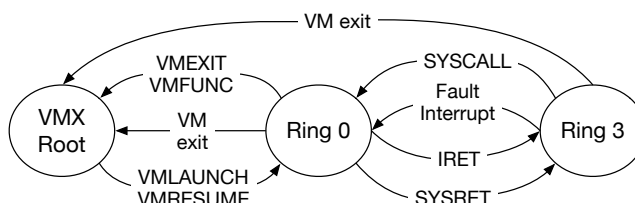
At the same time, less privileged software cannot be offered the ability to jump arbitrarily into more privileged code, as that would compromise the privileged software's ability to enforce security and isolation invariants. In our example, when an application wishes to write a file to the disk, the kernel must check if the application's user has access to that file. If the ring 3 code could perform an arbitrary jump in kernel space, it would be able to skip the access check.

For these reasons, the Intel architecture includes privilege-switching mechanisms used to transfer control from less privileged software to well-defined entry points in more privileged software. As suggested above, an architecture's privilege-switching mechanisms have deep implications for the security properties of its software. Furthermore, securely executing the software inside a protected container requires the same security considerations as privilege level switching.

Due to historical factors, the Intel architecture has a vast number of execution modes, and an intimidating amount of transitions between them. We focus on the privilege level switching mechanisms used by modern 64-bit software, summarized in Figure 2.16.

### **2.8.1 System Calls**

On modern processors, application software uses the `SYSCALL` instruction to invoke ring 0 code, and the kernel uses `SYSRET` to switch the privilege level back to ring 3. `SYSCALL` jumps into a predefined kernel location, which is specified by writing to a pair of architectural MSRs (§ 2.4).



**Figure 2.16:** Modern privilege switching methods in the 64-bit Intel architecture.

All MSRs can only be read or written by ring 0 code. This is a crucial security property, because it entails that application software cannot modify `SYSCALL`'s MSRs. If that was the case, a rogue application could abuse the `SYSCALL` instruction to execute arbitrary kernel code, potentially bypassing security checks.

The `SYSRET` instruction switches the current privilege level from ring 0 back to ring 3, and jumps to the address in `RCX`, which is set by the `SYSCALL` instruction. The `SYSCALL` / `SYSRET` pair does not perform any memory access, so it out-performs the Intel architecture's previous privilege switching mechanisms, which saved state on a stack. The design can get away without referencing a stack because kernel calls are not recursive.

### 2.8.2 Faults

The processor also performs a switch from ring 3 to ring 0 when a *hardware exception* occurs while executing application code. Some exceptions indicate bugs in the application, whereas other exceptions require kernel action.

A *general protection fault* (`#GP`) occurs when software attempts to perform a disallowed action, such as setting the `CR3` register from ring 3.

A *page fault* (`#PF`) occurs when address translation encounters a page table entry whose `P` flag is 0, or when the memory inside a page is accessed in way that is inconsistent with the access bits in the page table entry. For example, when ring 3 software accesses the memory inside a page whose `S` bit is set, the result of the memory access is `#PF`.

When a hardware exception occurs in application code, the CPU performs a ring switch, and calls the corresponding *exception handler*. For example, the `#GP` handler typically terminates the application's process, while the `#PF` handler reads the swapped out page back into RAM and resumes the application's execution.

The exception handlers are a part of the OS kernel, and their locations are specified in the first 32 entries of the Interrupt Descriptor Table (IDT), whose structure is shown in Table 2.3. The IDT's physical address is stored in the IDTR register, which can only be accessed by ring 0 code. Kernels protect the IDT memory using page tables, so that ring 3 software cannot access it.

**Table 2.3:** The essential fields of an IDT entry in 64-bit mode. Each entry points to a hardware exception or interrupt handler.

| Field                             | Bits |
|-----------------------------------|------|
| Handler RIP                       | 64   |
| Handler CS                        | 16   |
| Interrupt Stack Table (IST) index | 3    |

Each IDT entry has a 3-bit index pointing into the Interrupt Stack Table (IST), which is an array of 8 stack pointers stored in the TSS described in § 2.7.

When a hardware exception occurs, the execution state may be corrupted, and the current stack cannot be relied on. Therefore, the CPU first uses the handler's IDT entry to set up a known good stack. SS is loaded with a null descriptor, and RSP is set to the IST value to which the IDT entry points. After switching to a reliable stack, the CPU pushes the snapshot in Table 2.4 on the stack, then loads the IDT entry's values into the CS and RIP registers, which trigger the execution of the exception handler.

After the exception handler completes, it uses the `IRET` (interrupt return) instruction to load the registers from the on-stack snapshot and switch back to ring 3.

The Intel architecture gives the fault handler complete control over the execution context of the software that incurred the fault. This privilege is necessary for handlers (e.g., `#GP`) that must perform context



**Table 2.4:** The snapshot pushed on the handler’s stack when a hardware exception occurs. IRET restores registers from this snapshot.

| Field          | Bits |
|----------------|------|
| Exception SS   | 64   |
| Exception RSP  | 64   |
| RFLAGS         | 64   |
| Exception CS   | 64   |
| Exception RIP  | 64   |
| Exception code | 64   |

switches (§ 2.6) as a consequence of terminating a thread that encountered a bug. It follows that all fault handlers must be trusted to not leak or tamper with the information in an application’s execution context.

### 2.8.3 VMX Privilege Level Switching

Intel systems that take advantage of the hardware virtualization support to host multiple operating systems concurrently use a hypervisor to manage the VMs. The hypervisor creates a *Virtual Machine Control Structure* (VMCS) for each operating system instance that it wishes to run, and uses the `VMENTER` instruction to assign a logical processor to the VM.

When a logical processor encounters a fault that must be handled by the hypervisor, the logical processor performs a VM exit. For example, if the address translation process encounters an EPT entry with the P flag set to 0, the CPU performs a VM exit, and the hypervisor has an opportunity to bring the page into RAM.

The VMCS shows a great application of the encapsulation principle [Liskov and Zilles, 1974], which is generally used in high-level software, to computer architecture. The Intel architecture specifies that each VMCS resides in DRAM and is 4 KB in size. However, the architecture does not specify the VMCS format, and instead requires the hypervisor to interact with the VMCS via CPU instructions such as `VMREAD` and `VMWRITE`.

This approach allows Intel to add VMX features that require VMCS format changes, without the burden of having to maintain backwards compatibility. This is no small feat, given that huge amounts of complexity in the Intel architecture were introduced due to compatibility requirements.

## 2.9 An Overview of a Modern Computer System

This section outlines the hardware components that make up a computer system based on the Intel architecture<sup>5</sup>.

§ 2.9.1 summarizes the structure of a *motherboard* relevant for a discussion of cost and impact of physical attacks against a computing system. § 2.9.2 describes Intel’s Management Engine, which plays a role in the computer’s bootstrap process, and has significant security implications. § 2.9.3 presents the major components of an Intel processor, and § 2.9.4 abstractly models an Intel execution core.

A thorough understanding of the above systems is instrumental not only for reasoning about physical attacks. More importantly, understanding the way resources are partitioned and shared by mutually distrusting parties is necessary to reason about software attacks based on information leakage, such as timing attacks.

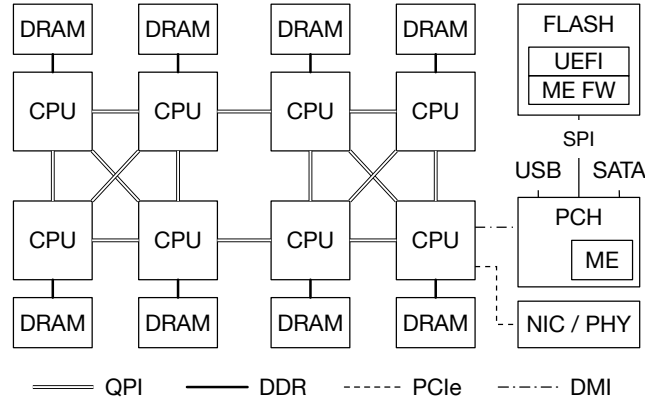
### 2.9.1 The Motherboard

A computer’s components are connected by a printed circuit board called a *motherboard*, shown in Figure 2.17, which consists of *sockets* connected by *buses*. Sockets connect chip-carrying *packages* to the board. The Intel documentation uses the term “package” to specifically refer to a CPU.

The CPU (described in § 2.9.3) hosts the execution cores that run the software stack shown in Figure 2.5 and described in § 2.3, namely the SMM code, the hypervisor, operating systems, and application processes. The computer’s main memory is provided by *Dynamic Random-Access Memory* (DRAM) modules.

---

<sup>5</sup>The information in here is drawn either from the SDM or in Intel’s Optimization Reference Manual [Int, 2014c].



**Figure 2.17:** The motherboard structures that are most relevant in a system security analysis.

The *Platform Controller Hub* (PCH) houses (relatively) low-speed I/O controllers driving the slower buses in the system, like SATA, used by storage devices, and USB, used by input peripherals. The PCH is also known as the *chipset*. At a first approximation, the *south bridge* term in older documentation can also be considered as a synonym for PCH.

Motherboards also have a non-volatile (flash) memory module storing firmware which implements the *Unified Extensible Firmware Interface* (UEFI) specification [UEF, 2015]. The firmware contains the boot code and the code that executes in System Management Mode (SMM, § 2.3).

The components we care about are connected by the following buses: the *Quick-Path Interconnect* (QPI [Int, 2010a]), a network of point-to-point links that connect processors, the *double data rate* (DDR) bus that connects a CPU to DRAM, the *Direct Media Interface* (DMI) bus that connects a CPU to the PCH, the *Peripheral Component Interconnect Express* (PCIe) bus that connects a CPU to peripherals such as a *Network Interface Card* (NIC), and the *Serial Programming Interface* (SPI) used by the PCH to communicate with the flash memory.

The PCIe bus is an extended, point-to-point version of the PCI standard, which provides a method for any peripheral connected to the bus to perform *Direct Memory Access* (DMA), transferring data to and from DRAM without involving an execution core and spending CPU cycles. The PCI standard includes a configuration mechanism that assigns a range of DRAM to each peripheral, but makes no provisions for restricting a peripheral's DRAM accesses to its assigned range.

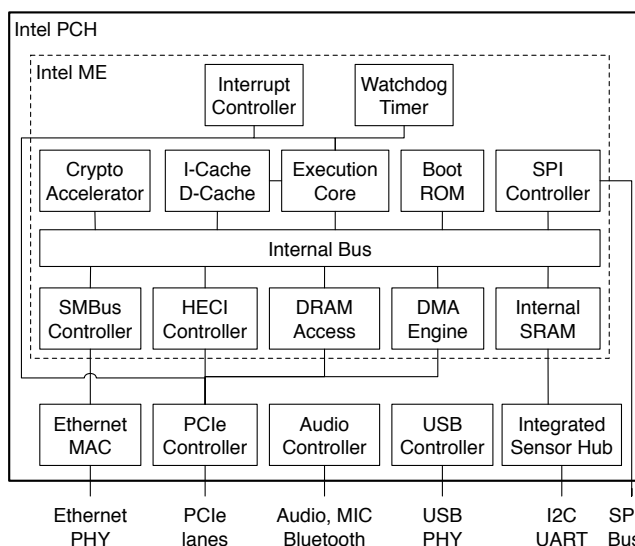
Network interfaces consist of a *physical* (PHY) module that converts the analog signals on the network media to and from digital bits, and a *Media Access Control* (MAC) module that implements a network-level protocol. Modern Intel-based motherboards forego a full-fledged NIC, and instead include an Ethernet [IEE, 2012] PHY module.

### 2.9.2 The Intel Management Engine (ME)

Intel's *Management Engine* (ME) is an embedded computer that was initially designed for remote system management and troubleshooting of server-class systems that are often hosted in data centers. However, all of Intel's recent PCHs contain an ME [Hofemeier, 2013], and it currently plays a crucial role in platform bootstrapping, which is described in detail in § 2.13. Most of the information in this section is obtained from an Intel-sponsored book [Ruan, 2014].

The ME is part of Intel's *Active Management Technology* (AMT), which is marketed as a convenient way for IT administrators to troubleshoot and fix situations such as failing hardware, or a corrupted OS installation, without having to gain physical access to the impacted computer.

The Intel ME, shown in Figure 2.18, remains functional during most hardware failures because it is an entire embedded computer featuring its own execution core, bootstrap ROM, and internal RAM. The ME can be used for troubleshooting effectively thanks to an array of abilities that include overriding the CPU's boot vector and a DMA engine that can access the computer's DRAM. The ME provides remote access to the computer without any CPU support because it can use the *System Management bus* (SMBus) to access the motherboard's Ethernet PHY or an AMT-compatible NIC [Int, 2015a].



**Figure 2.18:** The Intel Management Engine (ME) is an embedded computer hosted in the PCH. The ME has its own execution core, ROM and SRAM. The ME can access the host’s DRAM via a memory controller and a DMA controller. The ME is remotely accessible over the network, as it has direct access to an Ethernet PHY via the SMBus.

The Intel ME is connected to the motherboard’s power supply using a power rail that remains available even while the host computer is in the *Soft Off* mode [Int, 2015a] (otherwise known as ACPI G2/S5, where most of the computer’s components are powered off [Int, 2010d], including the CPU and DRAM). For all practical purposes, this means that the ME is active as long as the power supply is still connected to a power source.

In S5, the ME cannot access the DRAM, but it can use its own internal memories. The ME can also still communicate with a remote party, as it can access the motherboard’s Ethernet PHY via SMBus. This enables applications such as AMT’s theft prevention, where a laptop equipped with a cellular modem can be tracked and permanently disabled as long as it has power and is in range of a cellular network.

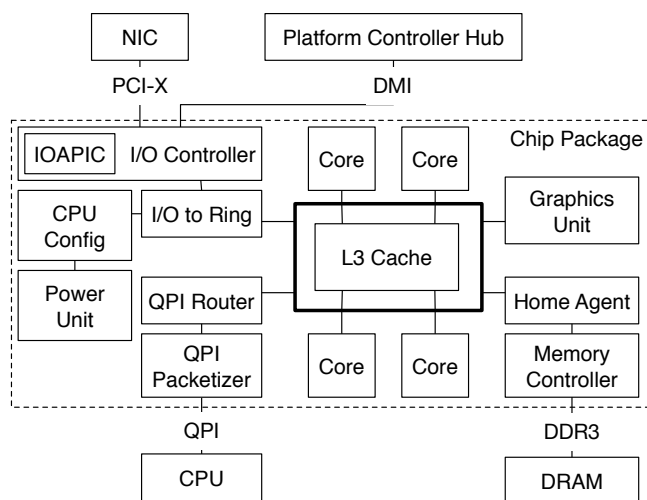
As the ME remains active in deep power-saving modes, its design must rely on low-power components. The execution core is an Argonaut

RISC Core (ARC) clocked at 200-400MHz, which is typically used in low-power embedded designs. On a very recent PCH [Int, 2015a], the internal SRAM has 640KB, and is shared with the Integrated Sensor Hub (ISH)’s core. The SMBus runs at 1MHz and, without CPU support, the motherboard’s Ethernet PHY runs at 10Mbps.

When the host computer is powered on, the ME’s processor begins executing code from the ME’s bootstrap ROM. The bootstrap code loads the ME’s software stack from the same flash module that stores the host computer’s firmware. The ME accesses the flash memory module an embedded SPI controller.

### 2.9.3 The Processor Die

An Intel processor’s die, illustrated in Figure 2.19, is divided into two broad areas: the *core area* implements the instruction execution pipeline typically associated with CPUs, while the *uncore* provides functions that were traditionally hosted in separate packages, but are currently integrated on the CPU die to reduce latency and power consumption.



**Figure 2.19:** The major components in a modern CPU package. § 2.9.3 gives an uncore overview. § 2.9.4 describes execution cores. § 2.11.3 takes a deeper look at the uncore.

At a conceptual level, the uncore of modern processors includes an *integrated memory controller* (iMC) that interfaces with the DDR bus, an *integrated I/O controller* (IIO) that implements PCIe bus lanes and interacts with the DMI bus, and a growing number of integrated peripherals, such as a *Graphics Processing Unit* (GPU). The uncore structure is described in some processor family datasheets [Int, 2014b,a], and in the overview sections in Intel’s uncore performance monitoring documentation [Corporation, 2014, Int, 2012b, 2010f].

Security extensions to the Intel architecture, such as Trusted Execution Technology (TXT) [Grawrock, 2009] and Software Guard Extensions (SGX) [McKeen et al., 2013, Anati et al., 2013], rely on the fact that the processor die includes the memory and I/O controller, and thus can prevent any device from accessing protected memory areas via *Direct Memory Access* (DMA) transfers. § 2.11.3 takes a deeper look at the uncore organization and at the machinery used to prevent unauthorized DMA transfers.

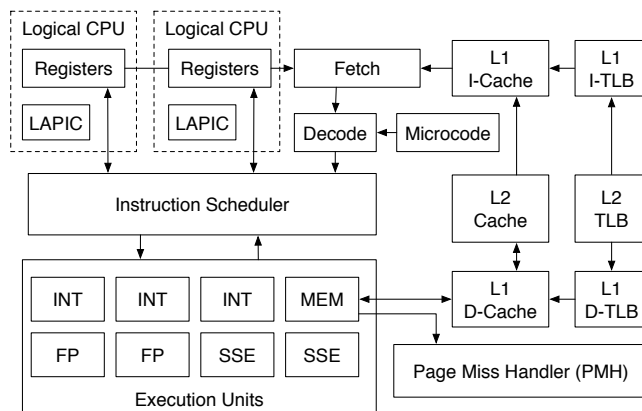
#### 2.9.4 The Core

Virtually all modern Intel processors have core areas consisting of multiple copies of the execution core circuitry, each of which is called a *core*. At the time of this writing, desktop-class Intel CPUs have 4 cores, and server-class CPUs have as many as 18 cores.

Most Intel CPUs feature *hyper-threading*, which means that a core (shown in Figure 2.20) has two copies of the register files backing the execution context described in § 2.6, and can execute two separate streams of instructions simultaneously. Hyper-threading reduces the impact of memory stalls on the utilization of the fetch, decode and execution units.

A hyper-threaded core is exposed to system software as two *logical processors* (LPs), also named *hardware threads* in the Intel documentation. The logical processor abstraction allows the code used to distribute work across processors in a multi-processor system to function without any change on multi-core hyper-threaded processors.

The high level of resource sharing introduced by hyper-threading introduces a security vulnerability. Software running on one logical pro-



**Figure 2.20:** CPU core with two logical processors. Each logical processor has its own execution context and LAPIC (§ 2.12). All other core resources are shared.

cessor can use the high-resolution performance counter (RDTSCP, § 2.4) [Petters and Farber, 1999] to get information about the instructions and memory access patterns of another piece of software that is executed on the other logical processor on the same core.

That being said, the biggest downside of hyper-threading may be the fact that writing about Intel processors in a rigorous manner requires the use of the cumbersome term Logical Processor instead of the shorter and more intuitive “CPU core”, or “core”.

## 2.10 Out-of-Order and Speculative Execution

CPU cores can execute instructions orders of magnitude faster than DRAM can read data. Computer architects attempt to bridge this gap by using hyper-threading (§ 2.9.3), out-of-order and speculative execution, and caching, which is described in § 2.11. In CPUs that use out-of-order execution, the order in which the CPU carries out a program’s instructions (*execution order*) is not necessarily the same as the order in which the instructions would be executed by a sequential evaluation system (*program order*).

An analysis of a system’s information leakage must take out-of-order execution into consideration. Any CPU actions observed by an



attacker match the execution order, so the attacker may learn some information by comparing the observed execution order with a known program order. At the same time, attacks that try to infer a victim's program order based on actions taken by the CPU must account for out-of-order execution as a source of noise.

This section summarizes the out-of-order and speculative execution concepts used when reasoning about a system's security properties. [Patterson and Hennessy, 2013] and [Hennessy and Patterson, 2012] cover the concepts in great depth, while Intel's optimization manual [Int, 2014c] provides details specific to Intel CPUs.

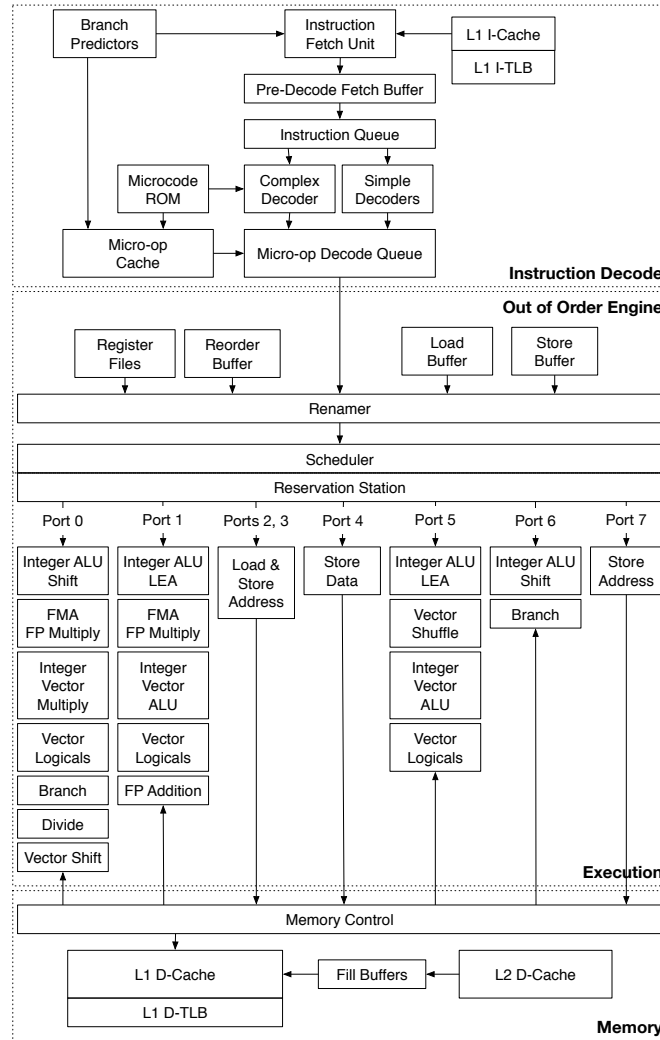
Figure 2.21 provides a more detailed view of the CPU core components involved in out-of-order execution, and omits some less relevant details from Figure 2.20.

The Intel architecture defines a *complex instruction set* (CISC). However, virtually all modern CPUs are architected following *reduced instruction set* (RISC) principles. This is accomplished by having the instruction decode stages break down each instruction into *micro-ops*, which resemble RISC instructions. The other stages of the execution pipeline work exclusively with micro-ops.

### 2.10.1 Out-of-Order Execution

Different types of instructions require different logic circuits, called *functional units*. For example, the arithmetic logic unit (ALU), which performs arithmetic operations, is completely different from the load and store unit, which performs memory operations. Different circuits can be used at the same time, so each CPU core can execute multiple micro-ops in parallel.

The core's out-of-order engine receives decoded micro-ops, identifies the micro-ops that can execute in parallel, assigns them to functional units, and combines the outputs of the units so that the results are equivalent to having the micro-ops executed sequentially in the order in which they come from the decode stages.



**Figure 2.21:** The structures in a CPU core that are relevant to out-of-order and speculative execution. Instructions are decoded into micro-ops, which are scheduled on one of the execution unit's ports. The branch predictor enables speculative execution when a branch is encountered.

For example, consider the sequence of pseudo micro-ops<sup>6</sup> in Table 2.5 below. The **OR** uses the result of the **LOAD**, but the **ADD** does not. Therefore, a good scheduler can have the load store unit execute the **LOAD** and the ALU execute the **ADD**, all in the same clock cycle.

**Table 2.5:** Pseudo micro-ops for the out-of-order execution example.

| # | Micro-op          | Meaning                           |
|---|-------------------|-----------------------------------|
| 1 | LOAD RAX, RSI     | $RAX \leftarrow \text{DRAM}[RSI]$ |
| 2 | OR RDI, RDI, RAX  | $RDI \leftarrow RDI \vee RAX$     |
| 3 | ADD RSI, RSI, RCX | $RSI \leftarrow RSI + RCX$        |
| 4 | SUB RBX, RSI, RDX | $RBX \leftarrow RSI - RDX$        |

The out-of-order engine in recent Intel CPUs works roughly as follows. Micro-ops received from the decode queue are written into a *reorder buffer* (ROB) while they are *in-flight* in the execution unit. The *register allocation table* (RAT) matches each register with the last reorder buffer entry that updates it. The *renamer* uses the RAT to rewrite the source and destination fields of micro-ops when they are written in the ROB, as illustrated in Tables 2.6 and 2.7. Note that the ROB representation makes it easy to determine the dependencies between micro-ops.

**Table 2.6:** Data written by the renamer into the reorder buffer (ROB), for the micro-ops in Table 2.5.

| # | Op   | Source 1 | Source 2    | Destination |
|---|------|----------|-------------|-------------|
| 1 | LOAD | RSI      | $\emptyset$ | RAX         |
| 2 | OR   | RDI      | ROB #1      | RSI         |
| 3 | ADD  | RSI      | RCX         | RSI         |
| 4 | SUB  | ROB # 3  | RDX         | RBX         |

The scheduler decides which micro-ops in the ROB get executed, and places them in the *reservation station*. The reservation station has one port for each functional unit that can execute micro-ops inde-

<sup>6</sup>The set of micro-ops used by Intel CPUs is not publicly documented. The fictional examples in this section suffice for illustration purposes.

**Table 2.7:** Relevant entries of the register allocation table after the micro-ops in Table 2.5 are inserted into the ROB.

| Register | RAX | RBX | RCX         | RDX         | RSI | RDI |
|----------|-----|-----|-------------|-------------|-----|-----|
| ROB #    | #1  | #4  | $\emptyset$ | $\emptyset$ | #3  | #2  |

pendently. Each reservation station port holds one micro-op from the ROB. The reservation station port waits until the micro-op's dependencies are satisfied and forwards the micro-op to the functional unit. When the functional unit completes executing the micro-op, its result is *written back* to the ROB, and forwarded to any other reservation station port that depends on it.

The ROB stores the results of completed micro-ops until they are *retired*, meaning that the results are *committed* to the register file and the micro-ops are removed from the ROB. Although micro-ops can be executed out-of-order, they must be retired in program order, in order to handle exceptions correctly. When a micro-op causes a hardware exception (§ 2.8.2), all of the following micro-ops in the ROB are *squashed*, and their results are discarded.

In the example above, the `ADD` can complete before the `LOAD`, because it does not require a memory access. However, the `ADD`'s result cannot be committed before `LOAD` completes. Otherwise, if the `ADD` is committed and the `LOAD` causes a page fault, software will observe an incorrect value for the `RSI` register.

The ROB is tailored for discovering register dependencies between micro-ops. However, micro-ops that execute out-of-order can also have memory dependencies. For this reason, out-of-order engines have a *load buffer* and a *store buffer* that keep track of in-flight memory operations and are used to resolve memory dependencies.

### 2.10.2 Speculative Execution

Branch control flow instructions, also called *branches*, change the instruction pointer (RIP, § 2.6), if a condition is met (*the branch is taken*). They implement conditional statements (`if`) and looping statements (such as `while` and `for`). The most well-known branching in-

structions in the Intel architecture are in the `jcc` family, such as `je` (jump if equal).

Branches pose a challenge to the decode stage, because the instruction that should be fetched after a branch is not known until the branching condition is evaluated. In order to avoid stalling the decode stage, modern CPU designs include *branch predictors* that use historical information to guess whether a branch will be taken or not.

When the decode stage encounters a branch instruction, it asks the branch predictor for a guess as to whether the branch will be taken or not. The decode stage bundles the branch condition and the predictor's guess into a branch check micro-op, and then continues decoding on the path indicated by the predictor. The micro-ops following the branch check are marked as *speculative*.

When the branch check micro-op is executed, the branch unit checks whether the branch predictor's guess was correct. If that is the case, the branch check is retired successfully. The scheduler handles *mispredictions* by squashing all micro-ops following the branch check, and by signaling the instruction decoder to flush the micro-op decode queue and start fetching the instructions that follow the correct branch.

Modern CPUs also attempt to predict memory read patterns, so they can *prefetch* the memory locations that are about to be read into the cache. Prefetching minimizes the latency of successfully predicted read operations, as their data will already be cached. This is accomplished by exposing circuits called prefetchers to memory accesses and cache misses. Each prefetcher can recognize a particular access pattern, such as sequentially reading an array's elements. When memory accesses match the pattern that a prefetcher was built to recognize, the prefetcher loads the cache line corresponding to the next memory access in its pattern.

## 2.11 Memory Cache Subsystem

At the time of this writing, CPU cores can process data  $\approx 200\times$  faster than DRAM can supply it. This gap is bridged by an hierarchy of cache memory modules, which are orders of magnitude smaller and an order

of magnitude faster than DRAM. While caching is transparent to application software, the system software is responsible for managing and coordinating the caches that store address translation (§ 2.5) results.

Caches impact the security of a software system in two ways. First, the Intel architecture relies on system software to manage address translation caches, which becomes an issue in a threat model where the system software is untrusted. Second, caches in the Intel architecture are shared by all software running on the computer. This opens up the way for *cache timing attacks*, an entire class of software attacks that rely on observing the time differences between accessing a cached memory location and an uncached memory location.

This section summarizes the caching concepts and implementation details needed to reason about both classes of security problems mentioned above. [Smith, 1982], [Patterson and Hennessy, 2013] and [Hennessy and Patterson, 2012] provide a good background on low-level cache implementation concepts. § 3.8 describes cache timing attacks.

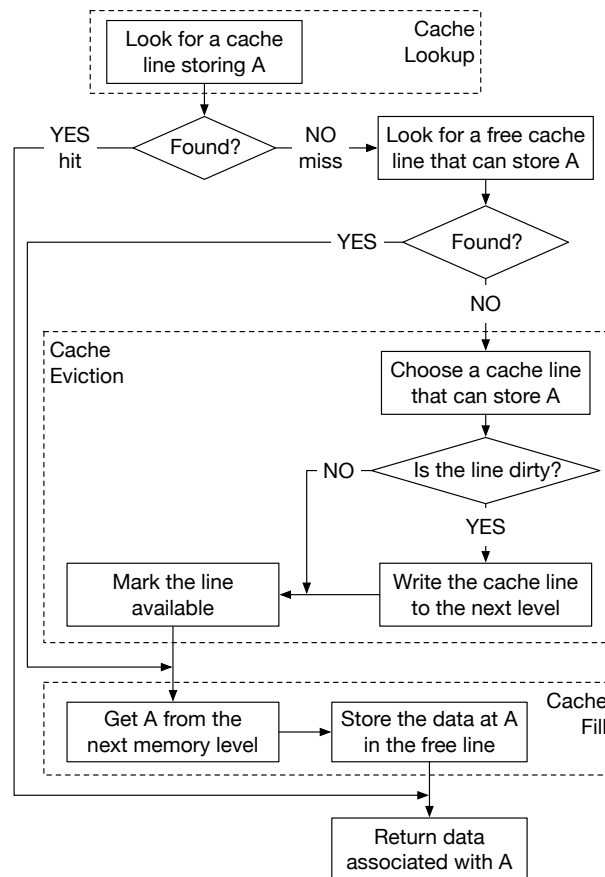
### 2.11.1 Caching Principles

At a high level, caches exploit the high locality in the memory access patterns of most applications to hide the main memory's (relatively) high latency. By *caching* (storing a copy of) the most recently accessed code and data, these relatively small memory structures can be used to satisfy 90%-99% of an application's memory accesses.

In an Intel processor, the *first-level* (L1) cache consists of a separate data cache (D-cache) and an instruction cache (I-cache). The instruction fetch and decode stage is directly connected to the L1 I-cache, and uses it to read the streams of instructions for the core's logical processors. Micro-ops that read from or write to memory are executed by the memory unit (MEM in Figure 2.20), which is connected to the L1 D-cache and forwards memory accesses to it.

Figure 2.22 illustrates the steps taken by a cache when it receives a memory access. First, a *cache lookup* uses the memory address to determine if the corresponding data exists in the cache. A *cache hit* occurs when the address is found, and the cache can resolve the memory access quickly. Conversely, if the address is not found, a *cache miss* occurs,

and a *cache fill* is required to resolve the memory access. When doing a fill, the cache forwards the memory access to the next level of the memory hierarchy and caches the response. Under most circumstances, a cache fill also triggers a *cache eviction*, in which some data is removed from the cache to make room for the data coming from the fill. If the data that is evicted has been modified since it was loaded in the cache, it must be *written back* to the next level of the memory hierarchy.



**Figure 2.22:** The steps taken by a cache memory to resolve an access to a memory address A. A normal memory access (to cacheable DRAM) always triggers a cache lookup. If the access misses the cache, a fill is required, and a write-back may be required.

Table 2.8 shows the key characteristics of the memory hierarchy implemented by modern Intel CPUs. Each core has its own L1 and L2 cache (see Figure 2.20), while the L3 cache is in the CPU’s uncore (see Figure 2.19), and is shared by all cores in the package.

**Table 2.8:** Approximate sizes and access times for each level in the memory hierarchy of an Intel processor, from [Levinthal, 2010]. Memory sizes and access times differ by orders of magnitude across the different levels of the hierarchy. This table does not cover multi-processor systems.

| Memory         | Size   | Access Time  |
|----------------|--------|--------------|
| Core Registers | 1 KB   | no latency   |
| L1 D-Cache     | 32 KB  | 4 cycles     |
| L2 Cache       | 256 KB | 10 cycles    |
| L3 Cache       | 8 MB   | 40-75 cycles |
| DRAM           | 16 GB  | 60 ns        |

The numbers in Table 2.8 suggest that cache placement can have a large impact on an application’s execution time. Because of this, the Intel architecture includes an assortment of instructions that give performance-sensitive applications some control over the caching of their working sets. **PREFETCH** instructs the CPU’s prefetcher to cache a specific memory address, in preparation for a future memory access. The memory writes performed by the **MOVNT** instruction family bypass the cache if a fill would be required. **CLFLUSH** evicts any cache lines storing a specific address from the entire cache hierarchy.

The methods mentioned above are available to software running at all privilege levels, because they were designed for high-performance workloads with large working sets, which are usually executed at ring 3 (§ 2.3). For comparison, the instructions used by system software to manage the address translation caches, described in § 2.11.5 below, can only be executed at ring 0.

### 2.11.2 Cache Organization

In the Intel architecture, caches are completely implemented in hardware, meaning that the software stack has no direct control over the eviction process. However, software can gain some control over which



data gets evicted by understanding how the caches are organized, and by cleverly placing its data in memory.

The *cache line* is the atomic unit of cache organization. A cache line has *data*, a copy of a continuous range of DRAM, and a *tag*, identifying the memory address that the data comes from. Fills and evictions operate on entire lines.

The cache line size is the size of the data, and is always a power of two. Assuming  $n$ -bit memory addresses and a cache line size of  $2^l$  bytes, the lowest  $l$  bits of a memory address are an offset into a cache line, and the highest  $n - l$  bits determine the cache line that is used to store the data at the memory location. All recent processors have 64-byte cache lines.

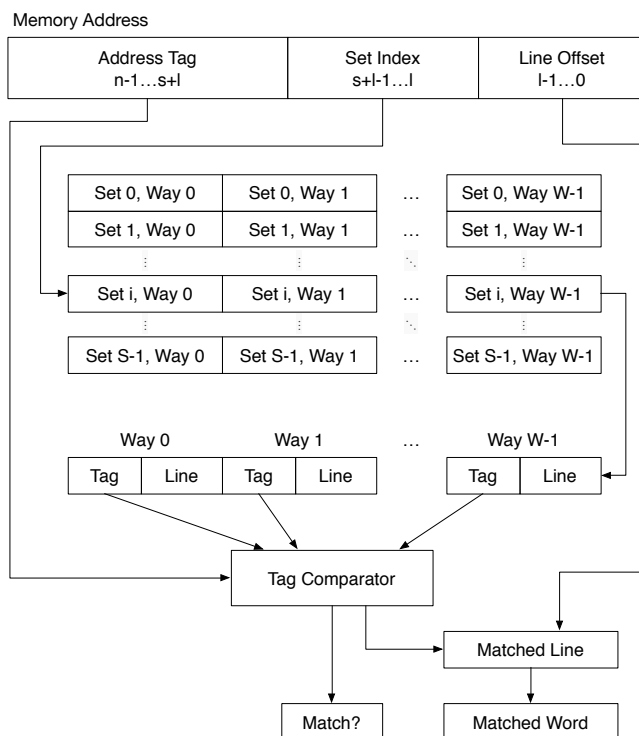
The L1 and L2 caches in recent processors are multi-way set-associative with direct set indexing, as shown in Figure 2.23. A  $W$ -way set-associative cache has its memory divided into *sets*, where each set has  $W$  lines. A memory location can be cached in any of the  $w$  lines in a specific set that is determined by the highest  $n - l$  bits of the location's memory address. Direct set indexing means that the  $S$  sets in a cache are numbered from 0 to  $S - 1$ , and the memory location at address  $A$  is cached in the set numbered  $A_{n-1...n-l} \bmod S$ .

In the common case where the number of sets in a cache is a power of two, so  $S = 2^s$ , the lowest  $l$  bits in an address make up the cache line offset, the next  $s$  bits are the set index. The highest  $n - s - l$  bits in an address are not used when selecting where a memory location will be cached. Figure 2.23 shows the cache structure and lookup process.

### 2.11.3 Cache Coherence

The Intel architecture was designed to support application software that was not written with caches in mind. One aspect of this support is the *Total Store Order* (TSO) [Owens et al., 2009] memory model, which promises that all logical processors in a computer see the same order of DRAM writes.

The same memory location may be simultaneously cached by different cores' caches, or even by caches on separate silicon dies, so providing the TSO guarantees requires a *cache coherence protocol*



**Figure 2.23:** Cache organization and lookup, for a  $W$ -way set-associative cache with  $2^l$ -byte lines and  $S = 2^s$  sets. The cache works with  $n$ -bit memory addresses. The lowest  $l$  address bits point to a specific byte in a cache line, the next  $s$  bytes index the set, and the highest  $n - s - l$  bits are used to decide if the desired address is in one of the  $W$  lines in the indexed set.

that synchronizes all cache lines in a computer that reference the same memory address.

The cache coherence mechanism is not visible to software, so it is only briefly mentioned in the SDM. Fortunately, Intel’s optimization reference [Int, 2014c] and the datasheets referenced in § 2.9.3 provide more information. Intel processors use variations of the MESIF [Goodman and Hum, 2009] protocol, which is implemented in the CPU and in the protocol layer of the QPI bus.

The SDM and the CPUID instruction output indicate that the L3 cache, also known as the *last-level cache* (LLC) is *inclusive*, meaning

that any location cached by an L1 or L2 cache must also be cached in the LLC. This design decision reduces complexity in many implementation aspects. We estimate that the bulk of the cache coherence implementation is in the CPU’s uncore, thanks to the fact that cache synchronization can be achieved without having to communicate to the lower cache levels that are inside execution cores.

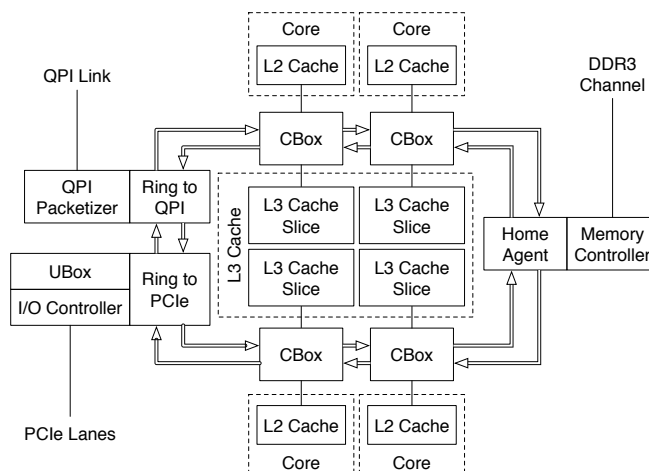
The QPI protocol defines *cache agents*, which are connected to the last-level cache in a processor, and *home agents*, which are connected to memory controllers. Cache agents make requests to home agents for cache line data on cache misses, while home agents keep track of cache line ownership, and obtain the cache line data from other cache line agents, or from the memory controller. The QPI routing layer supports multiple agents per socket, and each processor has its own caching agents, and at least one home agent.

Figure 2.24 shows that the CPU uncore has a bidirectional ring interconnect, which is used for communication between execution cores and the other uncore components. The execution cores are connected to the ring by *CBoxes*, which route their LLC accesses. The routing is static, as the LLC is divided into same-size slices (common slice sizes are 1.5 MB and 2.5 MB), and an undocumented hashing scheme maps each possible physical address to exactly one LLC slice.

Intel’s documentation states that the hashing scheme mapping physical addresses to LLC slices was designed to avoid having a slice become a hotspot, but stops short of providing any technical details. Fortunately, independent researches have reversed-engineered the hash functions for recent processors [Inci et al., 2015, Maurice et al., 2015, Yarom et al., 2015].

The hashing scheme described above is the reason why the L3 cache is documented as having a “complex” indexing scheme, as opposed to the direct indexing used in the L1 and L2 caches.

The number of LLC slices matches the number of cores in the CPU, and each LLC slice shares a CBox with a core. The CBoxes implement the cache coherence engine, so each CBox acts as the QPI cache agent for its LLC slice. CBoxes use a *Source Address Decoder* (SAD) to route DRAM requests to the appropriate home agents. Conceptu-



**Figure 2.24:** The stops on the ring interconnect used for inter-core and core-uncore communication.

ally, the SAD takes in a memory address and access type, and outputs a transaction type (coherent, non-coherent, IO) and a node ID. Each CBox contains a SAD replica, and the configurations of all SADs in a package are identical.

The SAD configurations are kept in sync by the *UBox*, which is the uncore configuration controller, and connects the *System agent* to the ring. The UBox is responsible for reading and writing physically distributed registers across the uncore. The UBox also receives interrupts from system and dispatches them to the appropriate core.

On recent Intel processors, the uncore also contains at least one memory controller. Each integrated memory controller (iMC or MBox in Intel’s documentation) is connected to the ring by a *home agent* (HA or *BBox* in Intel’s datasheets). Each home agent contains a *Target Address Decoder* (TAD), which maps each DRAM address to an address suitable for use by the DRAM modules, namely a DRAM channel, bank, rank, and a DIMM address. The mapping in the TAD is not documented by Intel, but it has been reverse-engineered [Pessl et al., 2015].

The integration of the memory controller on the CPU brings the ability to filter DMA transfers. Accesses from a peripheral connected

to the PCIe bus are handled by the integrated I/O controller (IIO), placed on the ring interconnect via the UBox, and then reach the iMC. Therefore, on modern systems, DMA transfers go through both the SAD and TAD, which can be configured to abort DMA transfers targeting protected DRAM ranges.

#### 2.11.4 Caching and Memory-Mapped Devices

Caches rely on the assumption that the underlying memory implements the memory abstraction in § 2.2. However, the physical addresses that map to memory-mapped I/O devices often deviate from the memory abstraction. For example, some devices expose command registers that trigger certain operations when written, and always return a zero value. Caching addresses that map to such memory-mapped I/O devices will lead to incorrect behavior.

Furthermore, even when the memory-mapped devices follow the memory abstraction, caching their memory is sometimes undesirable. For example, caching a graphic unit's frame buffer could lead to visual artifacts on the user's display, because of the delay between the time when a write is issued and the time when the corresponding cache lines are evicted and written back to memory.

In order to work around these problems, the Intel architecture implements a few caching behaviors, described below, and provides a method for partitioning the memory address space (§ 2.4) into regions, and for assigning a desired caching behavior to each region.

*Uncacheable* (UC) memory has the same semantics as the I/O address space (§ 2.4). UC memory is useful when a device's behavior is dependent on the order of memory reads and writes, such as in the case of memory-mapped command and data registers for a PCIe NIC (§ 2.9.1). The out-of-order execution engine (§ 2.10) does not reorder UC memory accesses, and does not issue speculative reads to UC memory.

*Write Combining* (WC) memory addresses the specific needs of frame buffers. WC memory is similar to UC memory, but the out-of-order engine may reorder memory accesses, and may perform speculative reads. The processor stores writes to WC memory in a write

combining buffer, and attempts to group multiple writes into a (more efficient) line write bus transaction.

*Write Through* (WT) memory is cached, but write misses do not cause cache fills. This is useful for preventing large rarely read memory-mapped device storage, such as frame buffers, from using cache memory. WT memory is covered by the cache coherence engine, may receive speculative reads, and is subject to operation reordering.

DRAM is represented as *Write Back* (WB) memory, which is optimized under the assumption that all devices that need to observe the memory operations implement the cache coherence protocol. WB memory is cached as described in § 2.11, receives speculative reads, and operations targeting it are subject to reordering.

*Write Protected* (WP) memory is similar to WB memory, with the exception that every write is propagated to the system bus. It is intended for memory-mapped buffers, where the order of operations does not matter, but the devices that need to observe the writes do not implement the cache coherence protocol, in order to reduce hardware costs.

On recent Intel processors, the cache's behavior is mainly configured by the *Memory Type Range Registers* (MTRRs) and by *Page Attribute Table* (PAT) indices in the page tables (§ 2.5). The behavior is also impacted by the Cache Disable (CD) and Not-Write through (NW) bits in Control Register 0 (CR0, § 2.4), as well as by equivalent bits in page table entries, namely Page-level Cache Disable (PCD) and Page-level Write-Through (PWT).

The MTRRs were intended to be configured by the computer's firmware during the boot sequence. Fixed MTRRs cover predetermined ranges of memory, such as the memory areas that had special semantics in the computers using 16-bit Intel processors. The ranges covered by *variable MTRRs* can be configured by system software. The representation used to specify the ranges is described below, as it has some interesting properties that have proven useful in other systems.

Each variable memory type range is specified using a *range base* and a *range mask*. A memory address belongs to the range if comput-

ing a bitwise AND between the address and the range mask results in the range base. This verification has a low-cost hardware implementation, shown in Figure 2.25.



**Figure 2.25:** The circuit for computing whether a physical address matches a memory type range. Assuming a CPU with 48-bit physical addresses, the circuit uses 36 AND gates and a binary tree of 35 XNOR (equality test) gates. The circuit outputs 1 if the address belongs to the range. The bottom 12 address bits are ignored, because memory type ranges must be aligned to 4 KB page boundaries.

Each variable memory type range must have a size that is an integral power of two, and a starting address that is a multiple of its size, so it can be described using the base / mask representation described above. A range’s starting address is its base, and the range’s size is one plus its mask.

Another advantage of this range representation is that the base and the mask can be easily validated, as shown in Listing 2.1. The range is aligned with respect to its size if and only if the bitwise AND between the base and the mask is zero. The range’s size is a power of two if and only if the bitwise AND between the mask and one plus the mask is zero. According to the SDM, the MTRRs are not validated, but setting them to invalid values results in undefined behavior.

---

```

constexpr bool is_valid_range(
    size_t base, size_t mask) {
    // Base is aligned to size.
    return (base & mask) == 0 &&
        // Size is a power of two.
        (mask & (mask + 1)) == 0;
}
  
```

---

**Listing 2.1:** The checks that validate the base and mask of a memory-type range can be implemented very easily.

No memory type range can partially cover a 4 KB page, which implies that the range base must be a multiple of 4 KB, and the bottom 12

bits of range mask must be set. This simplifies the interactions between memory type ranges and address translation, described in § 2.11.5.

The PAT is intended to allow the operating system or hypervisor to tweak the caching behaviors specified in the MTRRs by the computer's firmware. The PAT has 8 entries that specify caching behaviors, and is stored in its entirety in a MSR. Each page table entry contains a 3-bit index that points to a PAT entry, so the system software that controls the page tables can specify caching behavior at a very fine granularity.

### 2.11.5 Caches and Address Translation

Modern system software relies on address translation (§ 2.5). This means that all memory accesses issued by a CPU core use virtual addresses, which must undergo translation. Caches must know the physical address for a memory access, to handle aliasing (multiple virtual addresses pointing to the same physical address) correctly. However, address translation requires up to 20 memory accesses (see Figure 2.12), so it is impractical to perform a full address translation for every cache access. Instead, address translation results are cached in the *translation look-aside buffer* (TLB).

Table 2.9 shows the levels of the TLB hierarchy. Recent processors have separate L1 TLBs for instructions and data, and a shared L2 TLB. Each core has its own TLBs (see Figure 2.20). When a virtual address is not contained in a core's TLB, the *Page Miss Handler* (PMH) performs a *page walk* (page table / EPT traversal) to translate the virtual address, and the result is stored in the TLB.

**Table 2.9:** Approximate sizes and access times for each level in the TLB hierarchy, from [7zi, 2014].

| Memory      | Entries                          | Access Time       |
|-------------|----------------------------------|-------------------|
| L1 I-TLB    | $128 + 8 = 136$                  | 1 cycle           |
| L1 D-TLB    | $64 + 32 + 4 = 100$              | 1 cycle           |
| L2 TLB      | $1536 + 8 = 1544$                | 7 cycles          |
| Page Tables | $2^{36} \approx 6 \cdot 10^{10}$ | 18 cycles - 200ms |



In the Intel architecture, the PMH is implemented in hardware, so the TLB is never directly exposed to software and its implementation details are not documented. The SDM does state that each TLB entry contains the physical address associated with a virtual address, and the metadata needed to resolve a memory access. For example, the processor needs to check the writable (W) flag on every write, and issue a General Protection fault (#GP) if the write targets a read-only page. Therefore, the TLB entry for each virtual address caches the logical AND of all relevant W flags in the page table structures leading up to the page.

The TLB is transparent to application software. However, kernels and hypervisors must make sure that the TLBs do not get out of sync with the page tables and EPTs. When changing a page table or EPT, the system software must use the INVLPG instruction to invalidate any TLB entries for the virtual address whose translation changed. Some instructions *flush the TLBs*, meaning that they invalidate all TLB entries, as a side-effect.

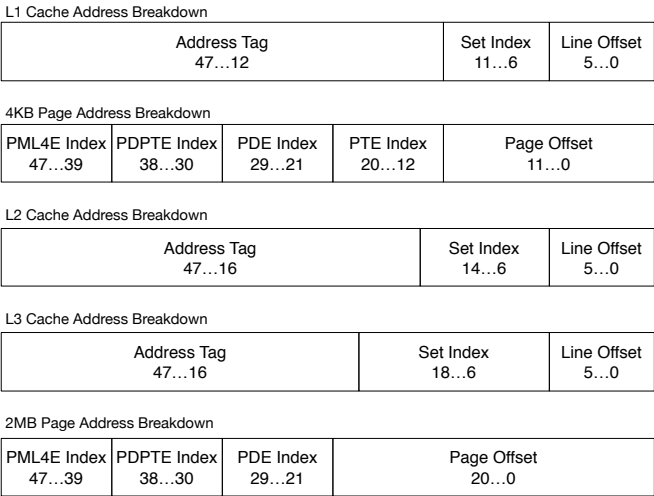
TLB entries also cache the desired caching behavior (§ 2.11.4) for their pages. This requires system software to flush the corresponding TLB entries when changing MTRRs or page table entries. In return, the processor only needs to compute the desired caching behavior during a TLB miss, as opposed to computing the caching behavior on every memory access.

The TLB is not covered by the cache coherence mechanism described in § 2.11.3. Therefore, when modifying a page table or EPT on a multi-core / multi-processor system, the system software is responsible for performing a *TLB shutdown*, which consists of stopping all logical processors that use the page table / EPT about to be changed, performing the changes, executing TLB-invalidating instructions on the stopped logical processors, and then resuming execution on the stopped logical processors.

Address translation constrains the L1 cache design. On Intel processors, the set index in an L1 cache only uses the address bits that are not impacted by address translation, so that the L1 set lookup can be

done in parallel with the TLB lookup. This is critical for achieving a low latency when both the L1 TLB and the L1 cache are hit.

Given a page size  $P = 2^p$  bytes, the requirement above translates to  $l + s \leq p$ . In the Intel architecture,  $p = 12$ , and all recent processors have 64-byte cache lines ( $l = 6$ ) and 64 sets ( $s = 6$ ) in the L1 caches, as shown in Figure 2.26. The L2 and L3 caches are only accessed if the L1 misses, so the physical address for the memory access is known at that time, and can be used for indexing.



**Figure 2.26:** Virtual addresses from the perspective of cache lookup and address translation. The bits used for the L1 set index and line offset are not changed by address translation, so the page tables do not impact L1 cache placement. The page tables do impact L2 and L3 cache placement. Using large pages (2 MB or 1 GB) is not sufficient to make L3 cache placement independent of the page tables, because of the LLC slice hashing function (§ 2.11.3).

## 2.12 Interrupts

Peripherals use *interrupts* to signal the occurrence of an event that must be handled by system software. For example, a keyboard triggers interrupts when a key is pressed or depressed. System software also relies on interrupts to implement preemptive multi-threading.

Interrupts are a kind of hardware exception (§ 2.8.2). Receiving an interrupt causes an execution core to perform a privilege level switch and to start executing the system software’s interrupt handling code. Therefore, the security concerns in § 2.8.2 also apply to interrupts, with the added twist that interrupts occur independently of the instructions executed by the interrupted code, whereas most faults are triggered by the actions of the application software that incurs them.

Given the importance of interrupts when assessing a system’s security, this section outlines the interrupt triggering and handling processes described in the SDM.

Peripherals use bus-specific protocols to signal interrupts. For example, PCIe relies on *Message Signaled Interrupts* (MSI), which are memory writes issued to specially designed memory addresses. The bus-specific interrupt signals are received by the *I/O Advanced Programmable Interrupt Controller* (IOAPIC) in the PCH, shown in Figure 2.17.

The IOAPIC routes interrupt signals to one or more *Local Advanced Programmable Interrupt Controllers* (LAPICs). As shown in Figure 2.19, each logical CPU has a LAPIC that can receive interrupt signals from the IOAPIC. The IOAPIC routing process assigns each interrupt to an 8-bit *interrupt vector* that is used to identify the interrupt sources, and to a 32-bit *APIC ID* that is used to identify the LAPIC that receives the interrupt.

Each LAPIC uses a 256-bit *Interrupt Request Register* (IRR) to track the unserved interrupts that it has received, based on the interrupt vector number. When the corresponding logical processor is available, the LAPIC copies the highest-priority unserved interrupt vector to the *In-Service Register* (ISR), and invokes the logical processor’s interrupt handling process.

At the execution core level, interrupt handling reuses many of the mechanisms of fault handling (§ 2.8.2). The interrupt vector number in the LAPIC’s ISR is used to locate an interrupt handler in the IDT, and the handler is invoked, possibly after a privilege switch is performed. The interrupt handler does the processing that the device requires, and

then writes the LAPIC's *End Of Interrupt* (EOI) register to signal the fact that it has completed handling the interrupt.

Interrupts are treated like faults, so interrupt handlers have full control over the execution environment of the application being interrupted. This is used to implement preemptive multi-threading, which relies on a clock device that generates interrupts periodically, and on an interrupt handler that performs context switches.

System software can cause an interrupt on any logical processor by writing the target processor's APIC ID into the *Interrupt Command Register* (ICR) of the LAPIC associated with the logical processor that the software is running on. These interrupts, called *Inter-Processor Interrupts* (IPI), are needed to implement TLB shoot-downs (§ 2.11.5).

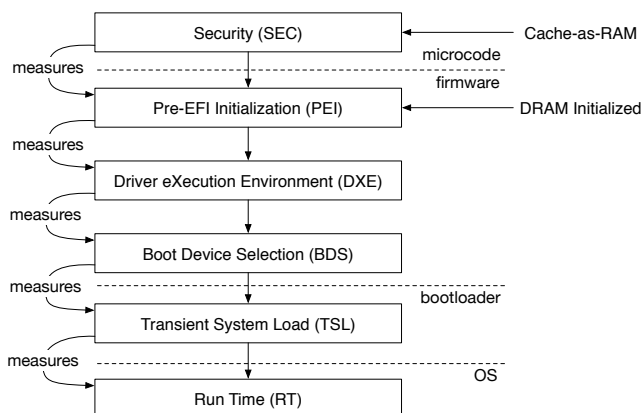
## 2.13 Platform Initialization (Bootting)

When a computer is powered up, it undergoes a *bootstrapping* process, also called *booting*, for simplicity. The boot process is a sequence of steps that collectively initialize all hardware components and load the system software into DRAM. An analysis of a system's security properties must be aware of all pieces of software executed during the boot process, and must account for the trust relationships that are created when a software module loads another module.

This section outlines the details of the boot process needed to reason about the security of a system based on the Intel architecture. [Int, 2010b] provides a good reference for many of the booting process's low-level details. While some specifics of the boot process depend on the motherboard and components in a computer, this section focuses on the high-level flow described by Intel's documentation.

### 2.13.1 The UEFI Standard

The firmware in recent computers with Intel processors implements the *Platform Initialization* (PI) process in the *Unified Extensible Firmware Interface* (UEFI) specification [UEF, 2015]. The platform initialization follows the steps shown in Figure 2.27 and described below.



**Figure 2.27:** The phases of the Platform Initialization process in the UEFI specification.

The computer powers up, reboots, or resumes from sleep in the *Security phase* (SEC). The SEC implementation is responsible for establishing a temporary memory store and loading the next stage of the firmware into it. As the first piece of software that executes on the computer, the SEC implementation is the system’s root of trust, and performs the first steps towards establishing the system’s desired security properties.

For example, in a measured boot system (also known as trusted boot), all software involved in the boot process is measured (cryptographically hashed, and the measurement is made available to third parties, as described in § 3.3). In such a system, the SEC implementation takes the first steps in establishing the system’s measurement, namely resetting the special register that stores the measurement result, measuring the PEI implementation, and storing the measurement in the special register.

SEC is followed by the *Pre-EFI Initialization phase* (PEI), which initializes the computer’s DRAM, copies itself from the temporary memory store into DRAM, and tears down the temporary storage. When the computer is powering up or rebooting, the PEI implementation is also responsible for initializing all non-volatile storage

units that contain UEFI firmware and loading the next stage of the firmware into DRAM.

PEI hands off control to the *Driver eXecution Environment phase* (DXE). In DXE, a loader locates and starts firmware drivers for the various components in the computer. DXE is followed by a Boot Device Selection (BDS) phase, which is followed by a Transient System Load (TSL) phase, where an EFI application loads the operating system selected in the BDS phase. Last, the OS loader passes control to the operating system's kernel, entering the Run Time (RT) phase.

When waking up from sleep, the PEI implementation first initializes the non-volatile storage containing the system snapshot saved while entering the sleep state. The rest of the PEI implementation may use optimized re-initialization processes, based on the snapshot contents. The DXE implementation also uses the snapshot to restore the computer's state, such as the DRAM contents, and then directly executes the operating system's wake-up handler.

### 2.13.2 SEC on Intel Platforms

Right after a computer is powered up, circuitry in the power supply and on the motherboard starts establishing reference voltages on the power rails in a specific order, documented as “power sequencing” [Venkataramani, 2011] in chipset specifications such as [Int, 2015h]. The rail powering up the Intel ME (§ 2.9.2) in the PCH is powered up significantly before the rail that powers the CPU cores.

When the ME is powered up, it starts executing the code in its boot ROM, which sets up the SPI bus connected to the flash memory module (§ 2.9.1) that stores both the UEFI firmware and the ME's firmware. The ME then loads its firmware from flash memory, which contains the ME's operating system and applications.

After the Intel ME loads its software, it sets up some of the motherboard's hardware, such as the PCH bus clocks, and then it kicks off the CPU's bootstrap sequence. Most of the details of the ME's involvement in the computer's boot process are not publicly available, but initializing the clocks is mentioned in a few public documents [Int, 2015b, pur, 2014, Dice, 2011, fit, 2014], and is made clear in firmware bringup

guides, such as the leaked confidential guide [Int, 2012a] documenting firmware bringup for Intel’s Series 7 chipset.

The beginning of the CPU’s bootstrap sequence is the SEC phase, which is implemented in the processor circuitry. All logical processors (LPs) on the motherboard undergo *hardware initialization*, which invalidates the caches (§ 2.11) and TLBs (§ 2.11.5), performs a *Built-In Self Test* (BIST), and sets all registers (§ 2.6) to pre-specified values.

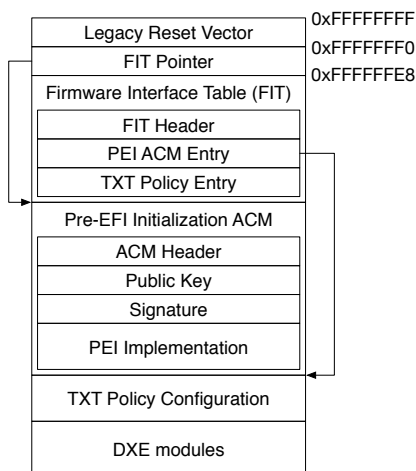
After hardware initialization, the LPs perform the Multi-Processor (MP) initialization algorithm, which results in one LP being selected as the *bootstrap processor* (BSP), and all other LPs being classified as *application processors* (APs).

According to the SDM, the details of the MP initialization algorithm for recent CPUs depend on the motherboard and firmware. In principle, after completing hardware initialization, all LPs attempt to issue a special no-op transaction on the QPI bus. A single LP will succeed in issuing the no-op, thanks to the QPI arbitration mechanism, and to the UBox (§ 2.11.3) in each CPU package, which also serves as a ring arbiter. The arbitration priority of each LP is based on its APIC ID (§ 2.12), which is provided by the motherboard when the system powers up. The LP that issues the no-op becomes the BSP. Upon failing to issue the no-op, the other LPs become APs, and enter the *wait-for-SIPI* state.

Understanding the PEI firmware loading process is unnecessarily complicated by the fact that the SDM describes a legacy process consisting of having the BSP set its RIP register to 0xFFFFFFFF0 (16 bytes below 4 GB), where the firmware is expected to place a instruction that jumps into the PEI implementation.

Recent processors do not support the legacy approach at all [Reinauer, 2013]. Instead, the BSP reads a word from address 0xFFFFF8 (24 bytes below 4 GB) [Zimmer and Yao, 2012, Datta and Kumar, 2013], and expects to find the address of a *Firmware Interface Table* (FIT) in the memory address space (§ 2.4), as shown in Figure 2.28. The BSP is able to read firmware contents from non-volatile memory before the computer is initialized, because the initial SAD (§ 2.11.3) and PCH

(§ 2.9.1) configurations maps a region in the memory address space to the SPI flash module (§ 2.9.1) that stores the computer’s firmware.



**Figure 2.28:** The Firmware Interface Table (FIT) in relation to the firmware’s memory map.

The FIT [Qureshi and Nicholes, 2006] was introduced in the context of Intel’s Itanium architecture, and its use in Intel’s current 64-bit architecture is described in an Intel patent [Datta and Kumar, 2013] and briefly documented in an obscure piece of TXT-related documentation [Int, 2010e]. The FIT contains *Authenticated Code Modules* (ACMs) that make up the firmware, and other platform-specific information, such as the TPM and TXT configuration [Int, 2010e].

The PEI implementation is stored in an ACM listed in the FIT. The processor loads the PEI ACM, verifies the trustworthiness of the ACM’s public key, and ensures that the ACM’s contents matches its signature. If the PEI passes the security checks, it is executed. Processors that support Intel TXT only accept Intel-signed ACMs [Futral and Greene, 2013, p. 92].

### 2.13.3 PEI on Intel Platforms

[Int, 2010b] and [Coreboot, 2014] describe the initialization steps performed by Intel platforms during the PEI phase, from the perspective



of a firmware programmer. A few steps provide useful context for reasoning about threat models involving the boot process.

When the BSP starts executing PEI firmware, DRAM is not yet initialized. Therefore, the PEI code starts executing in a *Cache-as-RAM* (CAR) mode, which only relies on the BSP's internal caches, at the expense of imposing severe constraints on the size of the PEI's working set.

One of the first tasks performed by the PEI implementation is enabling DRAM, which requires discovering and initializing the DRAM modules connected to the motherboard, and then configuring the BSP's memory controllers (§ 2.11.3) and MTRRs (§ 2.11.4). Most firmware implementations use Intel's *Memory Reference Code* (MRC) for this task.

After DRAM becomes available, the PEI code is copied into DRAM and the BSP is taken out of CAR mode. The BSP's LAPIC (§ 2.12) is initialized and used to send a broadcast *Startup Inter-Processor Interrupt* (SIPI, § 2.12) to wake up the APs. The interrupt vector in a SIPI indicates the memory address of the AP initialization code in the PEI implementation.

The PEI code responsible for initializing APs is executed when the APs receive the SIPI wake-up. The AP PEI code sets up the AP's configuration registers, such as the MTRRs, to match the BSP's configuration. Next, each AP registers itself in a system-wide table, using a memory synchronization primitive such as a semaphore, in order to avoid two APs accessing the table concurrently. After the AP initialization completes, each AP is suspended again, and waits to receive an INIT Inter-Processor Interrupt from the OS kernel.

The BSP initialization code waits for all APs to register themselves into the system-wide table, and then proceeds to locate, load and execute the firmware module that implements DXE.

## 2.14 CPU Microcode

The Intel architecture features a large instruction set. Some instructions are used infrequently, and some instructions are very complex,

which makes it impractical for an execution core to handle all instructions in hardware. Intel CPUs use a *microcode* table to break down rare and complex instructions into sequences of simpler instructions. Architectural extensions that only require microcode changes are significantly cheaper to implement and validate than extensions that require changes in the CPU's circuitry.

It follows that a good understanding of what can be done in microcode is crucial to evaluating the cost of security features that rely on architecture extensions. Furthermore, the limitations of microcode are sometimes the reasoning behind seemingly arbitrary architecture design decisions.

The first sub-section below presents the relevant facts pertaining to microcode in Intel's optimization reference [Int, 2014c] and SDM. The following subsections summarize information gleaned from Intel's patents and other researchers' findings.

### 2.14.1 The Role of Microcode

The frequently used instructions in the Intel architecture are handled by the core's fast path, which consists of simple decoders (§ 2.10) that can emit at most 4 micro-ops per instruction. Infrequently used instructions and instructions that require more than 4 micro-ops use a slower decoding path that relies on a sequencer to read micro-ops from a *microcode store ROM* (MSROM).

The 4 micro-ops limitation can be used to guess intelligently whether an architectural feature is implemented in microcode. For example, it is safe to assume that **XSAVE** (§ 2.6), which takes over 200 micro-ops on recent CPUs [Fog, 2014], is most likely performed in microcode, whereas simple arithmetic and memory accesses are handled directly by hardware.

The core's execution units handle common cases in fast paths implemented in hardware. When an input cannot be handled by the fast paths, the execution unit issues a *microcode assist*, which points the microcode sequencer to a routine in microcode that handles the edge cases. The most common cited example in Intel's documenta-

tion is floating point instructions, which `issue` assists to handle denormalized inputs.

The `REP MOVSB` family of instructions, also known as *string instructions* because of their use in `strcpy`-like functions, operate on variable-sized arrays. These instructions can handle small arrays in hardware, and `issue` microcode assists for larger arrays.

Modern Intel processors implement a microcode update facility. The SDM describes the process of applying microcode updates from the perspective of system software. Each core can be updated independently, and the updates must be reapplied on each boot cycle. A core can be updated multiple times. The latest SDM at the time of this writing states that a microcode update is up to 16 KB in size.

Processor engineers prefer to build new architectural features as microcode extensions, because microcode can be iterated on much faster than hardware, which reduces development cost [Wu and Bertin, 2008, Wu et al., 2012]. The update facility further increases the appeal of microcode, as some classes of bugs can be fixed after a CPU has been released.

Intel patents [McKeen et al., 2009, Johnson et al., 2010] describing Software Guard Extensions (SGX) disclose that SGX is entirely implemented in microcode, except for the memory encryption engine. A description of SGX's implementation could provide great insights into Intel's microcode, but, unfortunately, the SDM chapters covering SGX do not include such a description. We therefore rely on other public information sources about the role of microcode in the security-sensitive areas covered by previous sections, namely memory management (§ 2.5, § 2.11.5), the handling of hardware exceptions (§ 2.8.2) and interrupts (§ 2.12), and platform initialization (§ 2.13).

The use of microcode assists can be measured using the *Precise Event Based Sampling* (PEBS) feature in recent Intel processors. PEBS provides counters for the number of micro-ops coming from MSROM, including complex instructions and assists, counters for the numbers of assists associated with some micro-op classes (SSE and AVX stores and transitions), and a counter for assists generated by all other micro-ops.

The PEBS feature itself is implemented using microcode assists (this is implied in the SDM and confirmed by [Knauth and Ireland, 2014]) when it needs to write the execution context into a PEBS record. Given the wide range of features monitored by PEBS counters, we assume that all execution units in the core can issue microcode assists, which are performed at micro-op retirement. This finding is confirmed by an Intel patent [Boggs and Rodgers, 1997], and is supported by the existence of a PEBS counter for the “number of microcode assists invoked by hardware upon micro-op writeback.”

Intel’s optimization manual describes one more interesting assist, from a memory system perspective. SIMD masked loads (using `VMASKMOV`) read a series of data elements from memory into a vector register. A mask register decides whether elements are moved or ignored. If the memory address overlaps an invalid page (e.g., the P flag is 0, § 2.5), a microcode assist is issued, even if the mask indicates that no element from the invalid page should be read. The microcode checks whether the elements in the invalid page have the corresponding mask bits set, and either performs the load or issues a page fault.

The description of machine checks in the SDM mentions page assists and page faults in the same context. We assume that the page assists are issued in some cases when a TLB miss occurs (§ 2.11.5) and the PMH has to walk the page table. The following section develops this assumption and provides supporting evidence from Intel’s assigned patents and published patent applications.

### **2.14.2 Microcode Structure**

According to a 2013 Intel patent [Hughes et al., 2013], the avenues considered for implementing new architectural features are a completely microcode-based implementation, using existing micro-ops, a microcode implementation with hardware support, which would use new micro-ops, and a complete hardware implementation, using finite state machines (FSMs).

The main component of the MSROM is a table of micro-ops [Wu and Breternitz, 2008, Wu et al., 2012]. According to an example in a 2012 Intel patent [Wu et al., 2012], the table contains on the order

of 20,000 micro-ops, and a micro-op has about 70 bits. On embedded processors, like the Atom, microcode may be partially compressed [Wu and Breternitz, 2008, Wu et al., 2012].

The MSROM also contains an event ROM, which is an array of pointers to event handling code in the micro-ops table [Rodgers et al., 1999]. Microcode events are hardware exceptions, assists, and interrupts [Boggs and Rodgers, 1997, Papworth et al., 1999, Cornaby and Chaffin, 2007]. The processor described in a 1999 patent [Rodgers et al., 1999] has a 64-entry event table, where the first 16 entries point to hardware exception handlers and the other entries are used by assists.

The execution units can issue an assist or signal a fault by associating an event code with the result of a micro-op. When the micro-op is committed (§ 2.10), the event code causes the out-of-order scheduler to squash all micro-ops that are in-flight in the ROB. The event code is forwarded to the microcode sequencer, which reads the micro-ops in the corresponding event handler [Boggs and Rodgers, 1997, Papworth et al., 1999].

The hardware exception handling logic (§ 2.8.2) and interrupt handling logic (§ 2.12) is implemented entirely in microcode [Papworth et al., 1999]. Therefore, changes to this logic are relatively inexpensive to implement on Intel processors. This is rather fortunate, as the Intel architecture's standard hardware exception handling process requires that the fault handler is trusted by the code that encounters the exception (§ 2.8.2), and this assumption cannot be satisfied by a design where the software executing inside a secure container must be isolated from the system software managing the computer's resources.

The execution units in modern Intel processors support microcode procedures, via dedicated microcode call and return micro-ops [Cornaby and Chaffin, 2007]. The micro-ops manage a hardware data structure that conceptually stores a stack of microcode instruction pointers, and is integrated with out-of-order execution and hardware exceptions, interrupts and assists.

Asides from special micro-ops, microcode also employs special load and store instructions, which turn into special bus cycles, to issue commands to other functional units [Rodgers et al., 1997]. The memory

addresses in the special loads and stores encode commands and input parameters. For example, stores to a certain range of addresses flush specific TLB sets.

### 2.14.3 Microcode and Address Translation

Address translation (§ 2.5) is configured by CR3, which stores the physical address of the top-level page table, and by various bits in CR0 and CR4, all of which are described in the SDM. Writes to these control registers are implemented in microcode, which stores extra information in microcode-visible registers [George et al., 2009].

When a TLB miss (§ 2.11.5) occurs, the memory execution unit forwards the virtual address to the *Page Miss Handler* (PMH), which performs the page walk needed to obtain a physical address. In order to minimize the latency of a page walk, the PMH is implemented as a *Finite-State Machine* (FSM) [Hildesheim et al., 2014, Raikin et al., 2014]. Furthermore, the PMH fetches the page table entries from memory by issuing “stuffed loads”, which are special micro-ops that bypass the reorder buffer (ROB) and go straight to the memory execution units (§ 2.10), thus avoiding the overhead associated with out-of-order scheduling [Glew et al., 1997, Rodgers et al., 1997, Hildesheim et al., 2014].

The FSM in the PMH handles the fast path of the entire address translation process, which assumes no address translation fault (§ 2.8.2) occurs [Glew et al., 1996, 1997, Papworth et al., 1999, Rodgers et al., 1999], and no page table entry needs to be modified [Glew et al., 1997].

When the PMH FSM detects the conditions that trigger a Page Fault or a General Protection Fault, it communicates a microcode event code, corresponding to the detected fault condition, to the execution unit (§ 2.10) responsible for memory operations [Glew et al., 1996, 1997, Papworth et al., 1999, Rodgers et al., 1999]. In turn, the execution unit triggers the fault by associating the event code with the micro-op that caused the address translation, as described in the previous section.

The PMH FSM does not set the Accessed or Dirty attributes (§ 2.5.3) in page table entries. When it detects that a page table entry must be modified, the FSM issues a microcode event

code for a page walk assist [Glew et al., 1997]. The microcode handler performs the page walk again, setting the A and D attributes on page table entries when necessary [Glew et al., 1997]. This finding was indirectly confirmed by the description for a PEBS event in the most recent SDM release.

The patents at the core of our descriptions above [Glew et al., 1996, Boggs and Rodgers, 1997, Glew et al., 1997, Papworth et al., 1999, Rodgers et al., 1999] were all issued between 1996 and 1999, which raises the concern of obsolescence. As Intel would not be able to file new patents for the same specifications, we cannot present newer patents with the information above. Fortunately, we were able to find newer patents that mention the techniques described above, proving their relevance to newer CPU models.

Two 2014 patents [Hildesheim et al., 2014, Raikin et al., 2014] mention that the PMH is executing a FSM which issues stuffing loads to obtain page table entries. A 2009 patent [George et al., 2009] mentions that microcode is invoked after a PMH walk, and that the microcode can prevent the translation result produced by the PMH from being written to the TLB.

A 2013 patent [Hughes et al., 2013] and a 2014 patent [Raikin and Valentine, 2014] on scatter / gather instructions disclose that the newly introduced instructions use a combination of hardware in the execution units that perform memory operations, which include the PMH. The hardware issues microcode assists for slow paths, such as gathering vector elements stored in uncacheable memory (§ 2.11.4), and operations that cause Page Faults.

A 2014 patent on APIC (§ 2.12) virtualization [Shanbhogue and Robinson, 2014] describes a memory execution unit modification that invokes a microcode assist for certain memory accesses, based on the contents of some range registers. The patent also mentions that the range registers are checked when the TLB miss occurs and the PMH is invoked, in order to decide whether a fast hardware path can be used for APIC virtualization, or a microcode assist must be issued.

The recent patents mentioned above allow us to conclude that the PMH in recent processors still relies on an FSM and stuffed loads,

and still uses microcode assists to handle infrequent and complex operations. This assumption plays a key role in estimating the implementation complexity of architectural modifications targeting the processor's address translation mechanism.

#### 2.14.4 Microcode and Booting

The SDM states that microcode performs the Built-In Self Test (BIST, § 2.13.2), but does not provide any details on the rest of the CPU's hardware initialization.

In fact, the entire SEC implementation on Intel platforms is contained in the processor microcode [Datta et al., 2010, Datta and Kumar, 2013, Shanbhogue and Robinson, 2014]. This implementation has desirable security properties, as it is significantly more expensive for an attacker to tamper with the MSROM circuitry (§ 2.14.2) than it is to modify the contents of the flash memory module that stores the UEFI firmware. § 3.4.3 and § 3.6 describe the broad classes of attacks that an Intel platform can be subjected to.

The microcode that implements SEC performs MP initialization (§ 2.13.2), as suggested in the SDM. The microcode then places the BSP into Cache-as-RAM (CAR) mode, looks up the PEI *Authenticated Code Module* (ACM) in the Firmware Interface Table (FIT), loads the PEI ACM into the cache, and verifies its signature (§ 2.13.2) [Datta et al., 2010, Zimmer and Robinson, 2012, Zimmer and Yao, 2012, Natu et al., 2012, Datta and Kumar, 2013]. Given the structure of ACM signatures, we can conclude that Intel's microcode contains implementations of RSA decryption and of a variant of SHA hashing.

The PEI ACM is executed from the CPU's cache, after it is loaded by the microcode [Datta et al., 2010, Zimmer and Robinson, 2012, Datta and Kumar, 2013]. This removes the possibility for an attacker with physical access to the SPI flash module to change the firmware's contents after the microcode computes its cryptographic hash, but before it is executed.

On motherboards compatible with LaGrande Server Extensions (LT-SX, also known as Intel TXT for servers), the firmware implementing PEI verifies that each CPU connected to motherboard supports



LT-SX, and powers off the CPU sockets that don't hold processors that implement LT-SX [Natu et al., 2012]. This prevents an attacker from tampering with a TXT-protected VM by hot-plugging a CPU in a running computer that is inside TXT mode. When a hot-plugged CPU passes security tests, a hypervisor is notified that a new CPU is available. The hypervisor updates its internal state, and sends the new CPU a SIPI. The new CPU executes a SIPI handler, inside microcode, that configures the CPU's state to match the state expected by the TXT hypervisor [Natu et al., 2012]. This implies that the AP initialization described in § 2.13.2 is implemented in microcode.

#### 2.14.5 Microcode Updates

The SDM explains that the microcode on Intel CPUs can be updated, and describes the process for applying an update. However, no detail about the contents of an update is provided. Analyzing Intel's microcode updates seems like a promising avenue towards discovering the structure of microcode system. Unfortunately, the updates have so far proven to be inscrutable [Chen and Ahn, 2014].

The microcode updates cannot be easily analyzed because they are encrypted, hashed with a cryptographic hash function like SHA-256, and signed using RSA or elliptic curve cryptography [Zimmer and Robinson, 2012]. The update facility is implemented entirely in microcode, including the decryption and signature verification [Zimmer and Robinson, 2012].

[Hawkes, 2012] independently used fault injection and timing analysis to conclude that each recent Intel microcode update is signed with a 2048-bit RSA key and a (possibly non-standard) 256-bit hash algorithm, which agrees with the findings above.

The microcode update implementation places the core's cache into No-Evict Mode (NEM, documented by the SDM) and copies the microcode update into the cache before verifying its signature [Zimmer and Robinson, 2012]. The update facility also sets up an MTRR entry to protect the update's contents from modifications via DMA transfers [Zimmer and Robinson, 2012] as it is verified and applied.

While Intel publishes the most recent microcode updates for each of its CPU models, the release notes associated with the updates are not publicly available. This is unfortunate, as the release notes could be used to confirm guesses that certain features are implemented in microcode.

However, some information can be inferred by reading through the Errata section in Intel’s Specification Updates [Int, 2010c, 2015d,e]. The phrase “it is possible for BIOS<sup>7</sup> to contain a workaround for this erratum” generally means that a microcode update was issued. For example, Errata AH in [Int, 2010c] implies that string instructions (`REP MOV`) are implemented in microcode, which was confirmed by Intel [Abraham, 2006].

Errata AH43 and AH91 in [Int, 2010c], and AAK73 in [Int, 2015d] imply that address translation (§ 2.5) is at least partially implemented in microcode. Errata AAK53, AAK63, and AAK70, AAK178 in [Int, 2015d], and BT138, BT210, in [Int, 2015e] imply that VM entries and exits (§ 2.8.2) are implemented in microcode, which is confirmed by the APIC virtualization patent [Shanbhogue and Robinson, 2014].

---

<sup>7</sup>Basic Input/Output System (BIOS) is the predecessor of UEFI-based firmware. Most Intel documentation, including the SDM, still uses the term BIOS to refer to firmware.

# 3

---

## A Primer on Security for Trusted Processors

---

Most systems rely on some cryptographic primitives for security. Unfortunately, these primitives employ many assumptions, and building a secure system by composing existing primitives is a very challenging endeavor. It follows that a system's security analysis should be particularly interested in what cryptographic primitives are used, and how they are integrated into the system.

§ 3.1 and § 3.2 lay the foundations for such an analysis by summarizing the primitives used by the secure architectures of interest to us, and by describing the most common constructs built using these primitives. § 3.3 builds on these concepts and describes software attestation, which is the most popular method for establishing trust in a secure architecture.

After describing the cryptographic foundations for building secure systems, we discuss the attacks that secure architectures must withstand. Besides from forming a security checklist for architecture design, these attacks build intuition for the design decisions in the architectures of interest to us.

The attacks that can be performed on a computer system are broadly classified into two general categories: software attacks and at-

tacks requiring physical access to the computer system. In *physical attacks*, the attacker compromises aspects of a system’s physical implementation to perform an operation that bypasses the limitations set by the computer system’s software abstraction layers. In contrast, *software attacks* are performed solely by executing software on the victim computer. § 3.4 summarizes the main types of physical attacks.

The distinction between software and physical attacks is particularly relevant in cloud computing scenarios, where gaining software access to the computer running a victim’s software can be accomplished with a credit card backed by modest funds [Ristenpart et al., 2009], whereas physical access is a more difficult prospect that requires trespass, coercion, or social engineering on the cloud provider’s employees.

However, the distinction between software and physical attacks is blurred by the attacks presented in § 3.6, which exploit programmable peripherals connected to the victim computer’s bus in order to carry out actions that are normally associated with physical attacks.

While the vast majority of software attacks exploit a bug in a software component, there are a few additional attack classes that deserve attention from architecture designers. Attacks exploiting the system’s virtual address translation mechanism, described in § 3.7, become relevant on architectures where the system software is not trusted. Cache timing attacks, summarized in § 3.8 exploit microarchitectural behaviors that are completely observable in software, but dismissed by the security analyses of most systems.

### 3.1 Cryptographic Primitives

This section overviews the cryptosystems used by secure architectures. We are interested in cryptographic primitives that guarantee confidentiality, integrity, and freshness, and we treat these primitives as black boxes, focusing on their use in larger systems. [Katz and Lindell, 2014] covers the mathematics behind cryptography, while [Ferguson et al., 2011] covers the topic of building systems out of cryptographic primitives. Tables 3.1 and 3.2 summarize the primitives covered in this section.

**Table 3.1:** Desirable security guarantees and primitives that provide them.

| Guarantee       | Primitive                 |
|-----------------|---------------------------|
| Confidentiality | <i>Encryption</i>         |
| Integrity       | <i>MAC / Signatures</i>   |
| Freshness       | <i>Nonces + integrity</i> |

**Table 3.2:** Popular cryptographic primitives that are considered to be secure against today's adversaries.

| Guarantee       | Symmetric Keys        | Asymmetric Keys          |
|-----------------|-----------------------|--------------------------|
| Confidentiality | AES-GCM,<br>AES-CTR   | RSA with<br>PKCS #1 v2.0 |
| Integrity       | HMAC-SHA-2<br>AES-GCM | DSS-RSA,<br>DSS-ECC      |

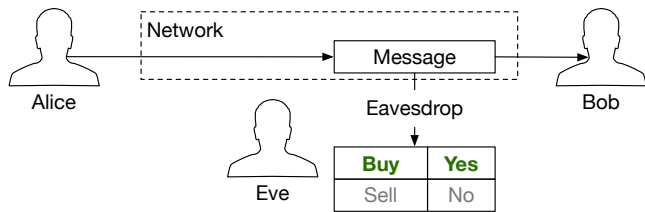
A message whose *confidentiality* is protected can be transmitted over an insecure medium without an adversary being able to obtain the information in the message. When *integrity* protection is used, the receiver is guaranteed to either obtain a message that was transmitted by the sender, or to notice that an attacker tampered with the message's content.

When multiple messages are transmitted over an untrusted medium, a *freshness* guarantee assures the receiver that she will obtain the latest message coming from the sender, or will notice an attack. A freshness guarantee is stronger than the equivalent integrity guarantee, because the latter does not protect against *replay attacks* where the attacker replaces a newer message with an older message coming from the same sender.

The following example further illustrates these concepts. Suppose Alice is a wealthy investor who wishes to either BUY or SELL an item every day. Alice cannot trade directly, and must relay her orders to her broker, Bob, over a network connection owned by Eve.

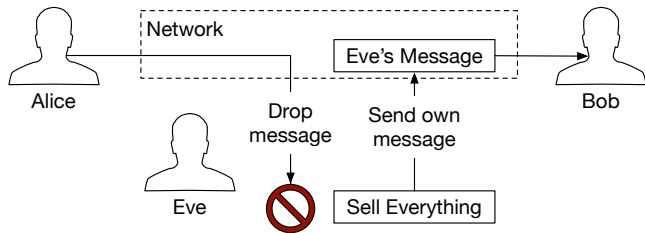
A communication system with confidentiality guarantees would prevent Eve from distinguishing between a BUY and a SELL order, as il-

lustrated in Figure 3.1. Without confidentiality, Eve would know Alice’s order before it is placed by Bob, so Eve would presumably gain a financial advantage at Alice’s expense.



**Figure 3.1:** In a confidentiality attack, Eve sees the message sent by Alice to Bob and can understand the information inside it. In this case, Eve can tell that the message is a **buy** order, and not a **sell** order.

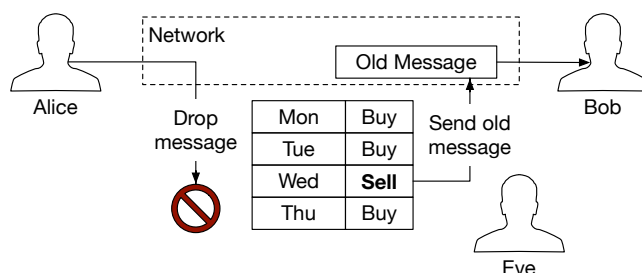
A system with integrity guarantees would prevent Eve from replacing Alice’s message with a false order, as shown in Figure 3.2. In this example, without integrity guarantees, Eve could replace Alice’s message with a SELL-EVERYTHING order, and buy Alice’s assets at a very low price.



**Figure 3.2:** In an integrity attack, Eve replaces Alice’s message with her own. In this case, Eve sends Bob a **sell-everything** order. In this case, Eve can tell that the message is a **buy** order, and not a **sell** order.

Last, a communication system that guarantees freshness would ensure that Eve cannot perform the replay attack pictured in Figure 3.3, where she would replace Alice’s message with an older message. Without freshness guarantees, Eve could mount the following attack, which bypasses both confidentiality and integrity guarantees. Over a few days, Eve would copy and store Alice’s messages from the network. When an order would reach Bob, Eve would observe the market and de-

termine if the order was BUY or SELL. After building up a database of messages labeled BUY or SELL, Eve would replace Alice’s message with an old message of her choice.



**Figure 3.3:** In a freshness attack, Eve replaces Alice’s message with a message that she sent at an earlier time. In this example, Eve builds a database of labeled messages over time, and is able to send Bob her choice of a BUY or a SELL order.

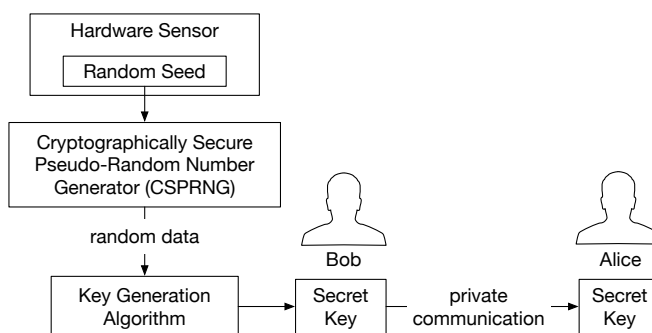
### 3.1.1 Cryptographic Keys

All cryptographic primitives that we describe here rely on *keys*, which are small pieces of information that must only be disclosed according to specific rules. A large part of a system’s security analysis focuses on ensuring that the keys used by the underlying cryptographic primitives are produced and handled according to the primitives’ assumptions.

Each cryptographic primitive has an associated *key generation algorithm* that uses random data to produce a unique key. The random data is produced by a *cryptographically strong pseudo-random number generator* (CSPRNG) that expands a small amount of *random seed* data into a much larger amount of data, which is computationally indistinguishable from true random data. The random seed must be obtained from a true source of randomness whose output cannot be predicted by an adversary, such as the least significant bits of the temperature readings coming from a hardware sensor.

*Symmetric key* cryptography requires that all parties in the system establish a shared *secret key*, which is usually referred to as “the key”. Typically, one party executes the key generation algorithm and securely transmits the resulting key to the other parties, as illustrated

in Figure 3.4. The channel used to distribute the key must provide confidentiality and integrity guarantees, which is a non-trivial logistical burden. The symmetric key primitives mentioned here do not make any assumption about the key, so the key generation algorithm simply grabs a fixed number of bits from the CSPRNG.



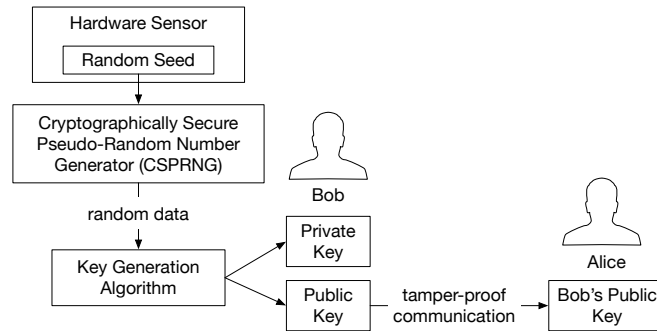
**Figure 3.4:** In symmetric key cryptography, a secret key is shared by the parties that wish to communicate securely.

The defining feature of *asymmetric key* cryptography is that it does not require a private channel for key distribution. Each party executes the key generation algorithm, which produces a *private key* and a *public key* that are mathematically related. Each party’s public key is distributed to the other parties over a channel with integrity guarantees, as shown in Figure 3.5. Asymmetric key primitives are more flexible than their symmetric counterparts, but are more complex and consume more computational resources.

### 3.1.2 Confidentiality

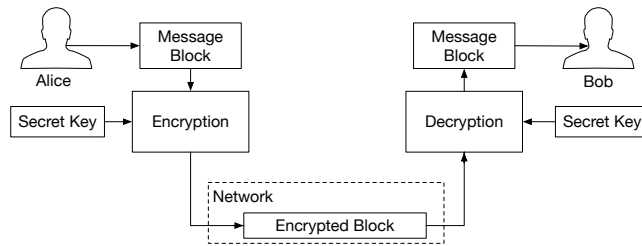
Many cryptosystems that provide integrity guarantees are built upon *block ciphers* that operate on fixed-size message blocks. The sender transforms a block using an *encryption* algorithm, and the receiver inverts the transformation using a *decryption* algorithm. The encryption algorithms in block ciphers obfuscate the message block’s content in the output, so that an adversary who does not have the decryption key cannot obtain the original message block from the encrypted output.





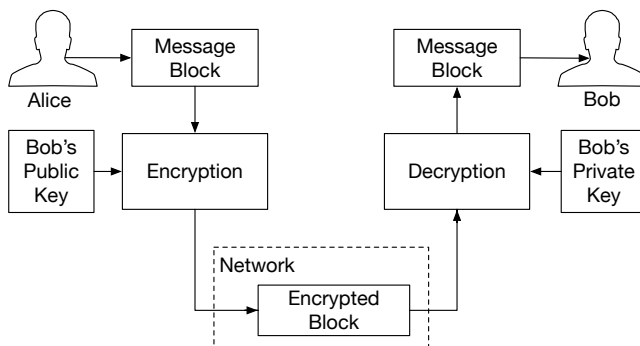
**Figure 3.5:** An asymmetric key generation algorithm produces a private key and an associated public key. The private key is held confidential, while the public key is given to any party who wishes to securely communicate with the private key's holder.

Symmetric key encryption algorithms use the same secret key for encryption and decryption, as shown in Figure 3.6, while asymmetric key block ciphers use the public key for encryption, and the corresponding private key for decryption, as shown in Figure 3.7.



**Figure 3.6:** In a symmetric key secure permutation (block cipher), the same secret key must be provided to both the encryption and the decryption algorithm.

The most popular block cipher based on symmetric keys at the time of this writing is the *American Encryption Standard* (AES) [Daemen and Rijmen, 1999, National Institute of Standards and Technology (NIST), 2001], with two variants that operate on 128-bit blocks using 128-bit keys or 256-bit keys. AES is a *secure permutation* function, as it can transform any 128-bit block into another 128-bit block. Recently, the United States *National Security Agency* (NSA) required the



**Figure 3.7:** In an asymmetric key block cipher, the encryption algorithm operates on a public key, and the decryption algorithm uses the corresponding private key.

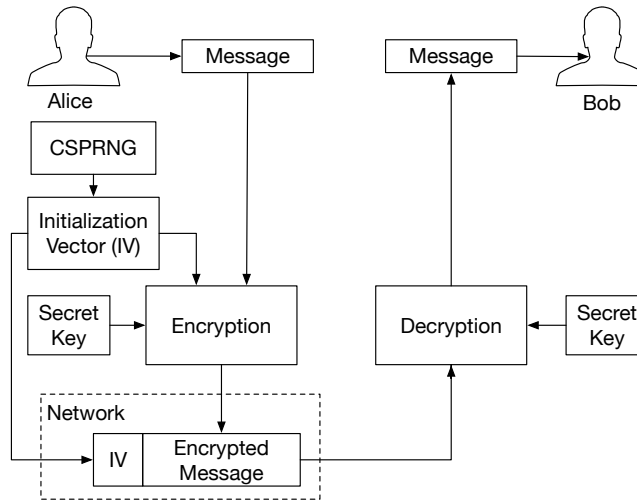
use of 256-bit AES keys for protecting sensitive information [National Security Agency (NSA) Central Security Service (CSS), 2015].

The most deployed asymmetric key block cipher is the *Rivest-Shamir-Adelman* (RSA) [Rivest et al., 1978] algorithm. RSA has variable key sizes, and 3072-bit key pairs are considered to provide the same security as 128-bit AES keys [Barker et al., 2012].

A block cipher does not necessarily guarantee confidentiality, when used on its own. A noticeable issue is that in our previous example, a block cipher would generate the same encrypted output for any of Alice’s BUY orders, as they all have the same content. Furthermore, each block cipher has its own assumptions that can lead to subtle vulnerabilities if the cipher is used directly.

Symmetric key block ciphers are combined with operating modes to form symmetric encryption schemes. Most operating modes require a random *initialization vector* (IV) to be used for each message, as shown in Figure 3.8. When analyzing the security of systems based on these cryptosystems, an understanding of the IV generation process is as important as ensuring the confidentiality of the encryption key.

Counter (CTR) and Cipher Block Chaining (CBC) are examples of operating modes recommended [Dworkin, 2001] by the United States *National Institute of Standards and Technology* (NIST), which informs the NSA’s requirements. Combining a block cipher, such as AES, with



**Figure 3.8:** Symmetric key block ciphers are combined with operating modes. Most operating modes require a random initialization vector (IV) to be generated for each encrypted message.

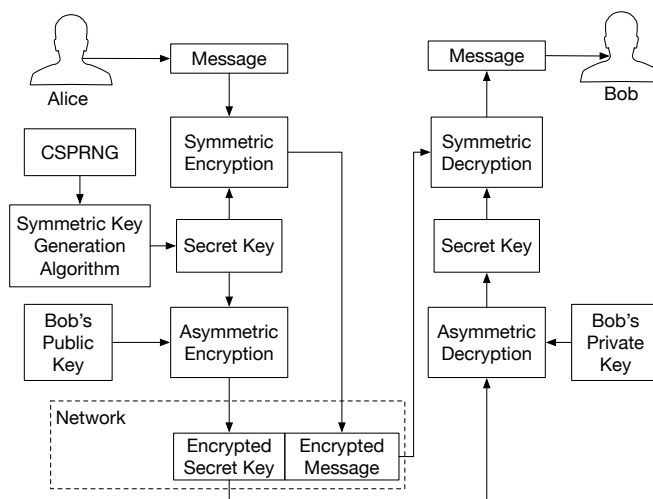
an operating mode, such as CTR, results in an encryption method, such as AES-CTR, which can be used to add confidentiality guarantees.

In the asymmetric key setting, there is no concept equivalent to operating modes. Each block cipher has its own assumptions, and requires a specialized scheme for general-purpose usage.

The RSA algorithm is used in conjunction with *padding methods*, the most popular of which are the methods described in the *Public-Key Cryptography Standard (PKCS) #1* versions 1.5 [Kaliski, 1998] and 2.0 [Kaliski and Staddon, 1998]. A security analysis of a system that uses RSA-based encryption must take the padding method into consideration. For example, the padding in PKCS #1 v1.5 can leak the private key under certain circumstances [Bleichenbacher, 1998]. While PKCS #1 v2.0 solves this issue, it is complex enough that some implementations have their own security issues [Manger, 2001].

Asymmetric encryption algorithms have much higher computational requirements than symmetric encryption algorithms. Therefore, when non-trivial quantities of data is encrypted, the sender generates a single-use secret key that is used to encrypt the data, and securely

communicates that secret key by encrypting it with the receiver’s public key, as shown in Figure 3.9.



**Figure 3.9:** Asymmetric key encryption is generally used to bootstrap a symmetric key encryption scheme.

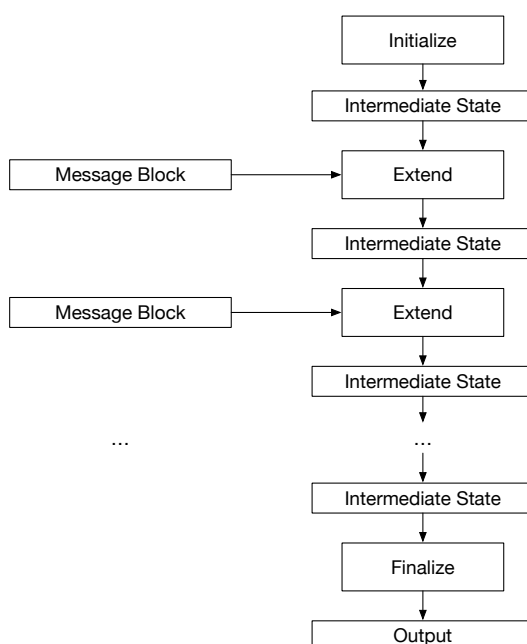
### 3.1.3 Integrity

Many cryptosystems that provide integrity guarantees are built upon *secure hashing* functions. These hash functions operate on an unbounded amount of input data and produce a small fixed-size output. Secure hash functions have a few guarantees, such as *pre-image resistance*, which states that an adversary cannot produce input data corresponding to a given hash output.

At the time of this writing, the most popular secure hashing function is the *Secure Hashing Algorithm* (SHA) [Eastlake and Jones, 2001]. However, due to security issues in SHA-1 [Stevens et al., 2015], new software is recommended to use at least 256-bit SHA-2 [Barker et al., 2015] for secure hashing.

The SHA hash functions are members of a large family of *block hash functions* that consume their input in fixed-size message blocks, and use a fixed-size internal state. A block hash function is used as

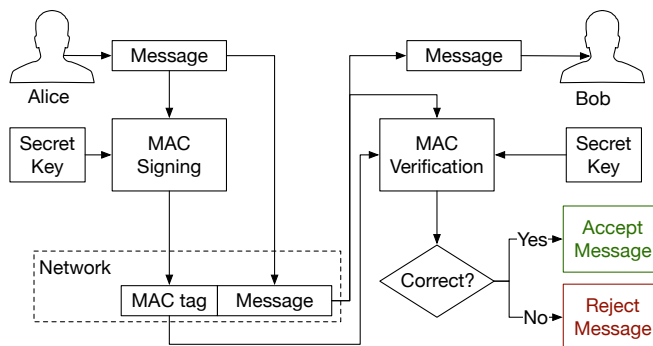
shown in Figure 3.10. An INITIALIZE algorithm is first invoked to set the internal state to its initial values. An EXTEND algorithm is executed for each message block in the input. After the entire input is consumed, a FINALIZE algorithm produces the hash output from the internal state.



**Figure 3.10:** A block hash function operates on fixed-size message blocks and uses a fixed-size internal state.

In the symmetric key setting, integrity guarantees are obtained using a *Message Authentication Code* (MAC) cryptosystem, illustrated in Figure 3.11. The sender uses a MAC algorithm that reads in a symmetric key and a variable-length message, and produces a fixed-length, short *MAC tag*. The receiver provides the original message, the symmetric key, and the MAC tag to a MAC verification algorithm that checks the authenticity of the message.

The key property of MAC cryptosystems is that an adversary cannot produce a MAC tag that will validate a message without the secret key.



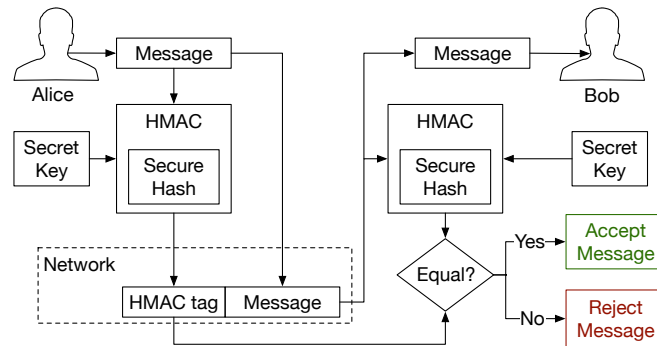
**Figure 3.11:** In the symmetric key setting, integrity is assured by computing a Message Authentication Code (MAC) tag and transmitting it over the network along the message. The receiver feeds the MAC tag into a verification algorithm that checks the message’s authenticity.

Many MAC cryptosystems do not have a separate MAC verification algorithm. Instead, the receiver checks the authenticity of the MAC tag by running the same algorithm as the sender to compute the expected MAC tag for the received message, and compares the output with the MAC tag received from the network.

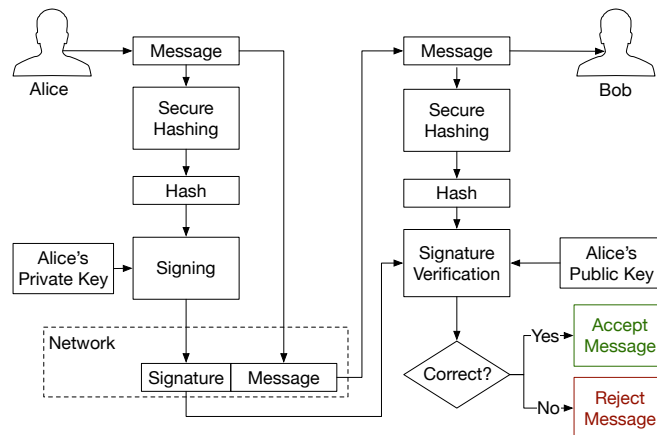
This is the case for the *Hash Message Authentication Code* (HMAC) [Krawczyk et al., 1997] generic construction, whose operation is illustrated in Figure 3.12. HMAC can use any secure hash function, such as SHA, to build a MAC cryptosystem.

Asymmetric key primitives that provide integrity guarantees are known as *signatures*. The message sender provides her private key to a *signing* algorithm, and transmits the output signature along with the message, as shown in Figure 3.13. The message receiver feeds the sender’s public key and the signature to a *signature verification* algorithm, which returns TRUE if the message matches the signature, and FALSE if the message has been tampered with.

Signing algorithms can only operate on small messages and are computationally expensive. Therefore, in practice, the message to be transmitted is first run through a cryptographically strong hash function, and the hash is provided as the input to the signing algorithm.



**Figure 3.12:** In the symmetric key setting, integrity is assured by computing a Hash-based Message Authentication Code (HMAC) and transmitting it over the network along the message. The receiver re-computes the HMAC and compares it against the version received from the network.



**Figure 3.13:** Signature schemes guarantee integrity in the asymmetric key setting. Signatures are created using the sender's private key, and are verified using the corresponding public key. A cryptographically secure hash function is usually employed to reduce large messages to small hashes, which are then signed.

At the time of this writing, the most popular choice for guaranteeing integrity in shared secret settings is HMAC-SHA, an HMAC function that uses SHA for hashing.

*Authenticated encryption*, which combines a block cipher with an operating mode that offers both confidentiality and integrity guarantees, is often an attractive alternative to HMAC. The most popular authenticated encryption operating mode is *Galois/Counter operation mode* (GCM) [McGrew and Viega, 2004], which has earned NIST’s recommendation [Dworkin, 2007] when combined with AES to form AES-GCM.

The most popular signature scheme combines the RSA encryption algorithms with padding schemes specified in PKCS #1, as illustrated in Figure 3.14. Recently, elliptic curve cryptography (ECC) [Koblitz, 1987] has gained a surge in popularity, thanks to its smaller key sizes. For example, a 384-bit ECC key is considered to be as secure as a 3072-bit RSA key [Barker et al., 2012, National Security Agency (NSA) Central Security Service (CSS), 2015]. The NSA requires the Digital Signature Standard (DSS) [National Institute of Standards and Technology (NIST), 2013], which specifies schemes based on RSA and ECC.

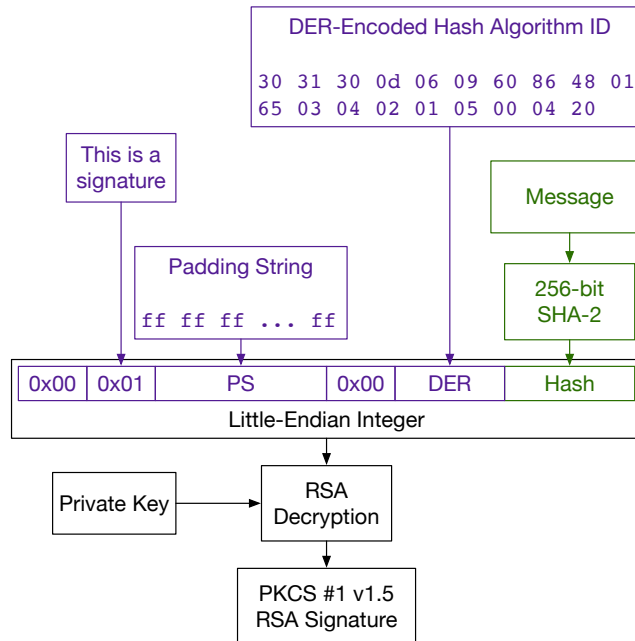
### 3.1.4 Freshness

Freshness guarantees are typically built on top of a system that already offers integrity guarantees, by adding a unique piece of information to each message. The main challenge in freshness schemes comes down to economically maintaining the trusted state needed to generate the unique pieces of information on the sender side, and verify their uniqueness on the receiver side.

A popular solution for gaining freshness guarantees relies on *nonces*, single-use random numbers. Nonces are attractive because the sender does not need to maintain any state; the receiver, however, must store the nonces of all received messages.

Nonces are often combined with a message timestamping and expiration scheme, as shown in Figure 3.15. An expiration can greatly reduce the receiver’s storage requirement, as the nonces for expired messages can be safely discarded. However, the scheme depends on

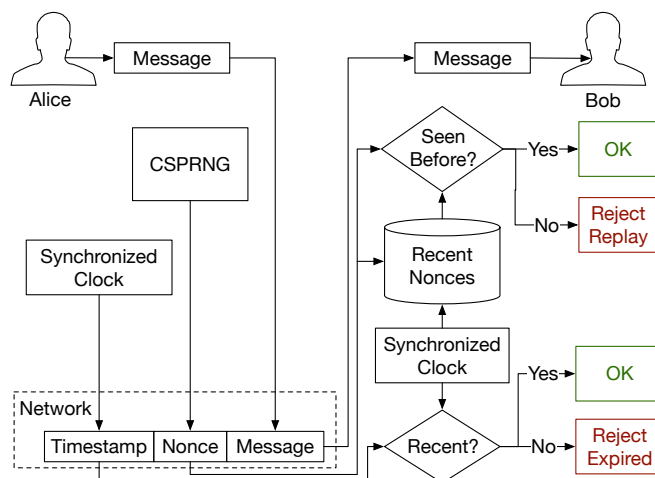




**Figure 3.14:** The RSA signature scheme with PKCS #1 v1.5 padding specified in RFC 3447 combines a secure hash of the signed message with a DER-encoded specification of the secure hash algorithm used by the signature, and a padding string whose bits are all set to 1. Everything except for the secure hash output is considered to be a part of the PKCS #1 v1.5 padding.

the sender and receiver having synchronized clocks. The message expiration time is a compromise between the desire to reduce storage costs, and the need to tolerate clock skew and delays in message transmission and processing.

Alternatively, nonces can be used in challenge-response protocols, in a manner that removes the storage overhead concerns. The challenger generates a nonce and embeds it in the challenge message. The response to the challenge includes an acknowledgment of the embedded nonce, so the challenger can distinguish between a fresh response and a replay attack. The nonce is only stored by the challenger, and is small in comparison to the rest of the state needed to validate the response.



**Figure 3.15:** Freshness guarantees can be obtained by adding timestamped nonces on top of a system that already offers integrity guarantees. The sender and the receiver use synchronized clocks to timestamp each message and discard unreasonably old messages. The receiver must check the nonce in each new message against a database of the nonces in all unexpired messages that it has seen.

## 3.2 Cryptographic Constructs

This section summarizes two constructs that are built on the cryptographic primitives described in § 3.1, and are used in the rest of this work.

### 3.2.1 Certificate Authorities

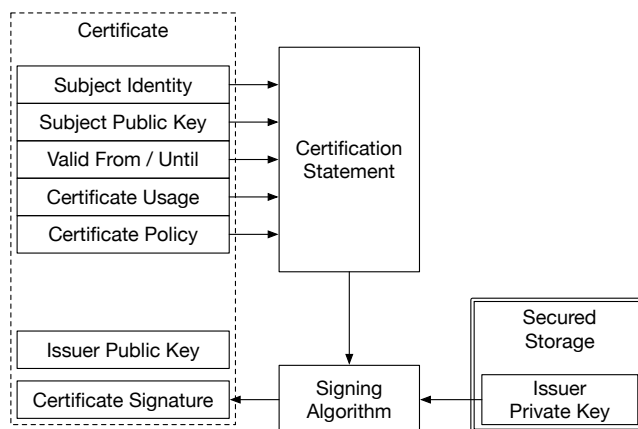
Asymmetric key cryptographic primitives assume that each party has the correct public keys for the other parties. This assumption is critical, as the entire security argument of an asymmetric key system rests on the fact that certain operations can only be performed by the owners of the private keys corresponding to the public keys. More concretely, if Eve can convince Bob that her own public key belongs to Alice, Eve can produce message signatures that seem to come from Alice.

The introductory material in § 3.1 assumed that each party transmits their public key over a channel with integrity guarantees. In prac-

tice, this is not a reasonable assumption, and the secure distribution of public keys is still an open research problem.

The most widespread solution to the public key distribution problem is the Certificate Authority (CA) system, which assumes the existence of a trusted authority whose public key is securely transmitted to all other parties in the system.

The CA is responsible for securely obtaining the public key of each party, and for issuing a *certificate* that binds a party's identity (e.g., "Alice") to its public key, as shown in Figure 3.16.



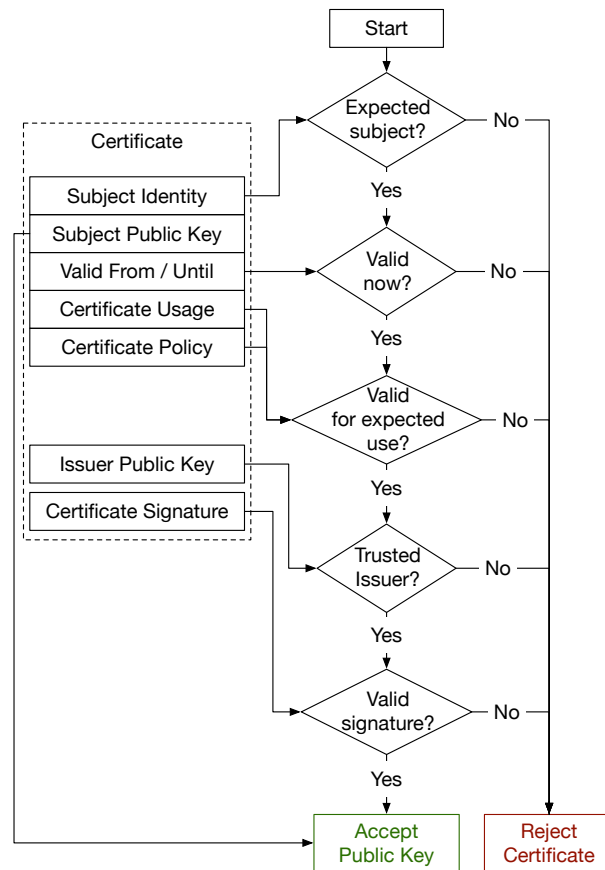
**Figure 3.16:** A certificate is a statement signed by a certificate authority (issuer) binding the identity of a subject to a public key.

A certificate is essentially a cryptographic signature produced by the private key of the certificate's *issuer*, who is generally a CA. The message signed by the issuer states that a public key belongs to a *subject*. The certificate message generally contains identifiers that state the intended use of the certificate, such as "the key in this certificate can only be used to sign e-mail messages". The certificate message usually also includes an identifier for the issuer's *certification policy*, which summarizes the means taken by the issuer to ensure the authenticity of the subject's public key.

A major issue in a CA system is that there is no obvious way to revoke a certificate. A revocation mechanism is desirable to handle situations where a party's private key is accidentally exposed, to

avoid having an attacker use the certificate to impersonate the compromised party. While advanced systems for certificate revocation have been developed, the first line of defense against key compromise is adding expiration dates to certificates.

In a CA system, each party presents its certificate along with its public key. Any party that trusts the CA and has obtained the CA's public key securely can verify any certificate using the process illustrated in Figure 3.17.



**Figure 3.17:** A certificate issued by a CA can be validated by any party that has securely obtained the CA's public key. If the certificate is valid, the subject public key contained within can be trusted to belong to the subject identified by the certificate.

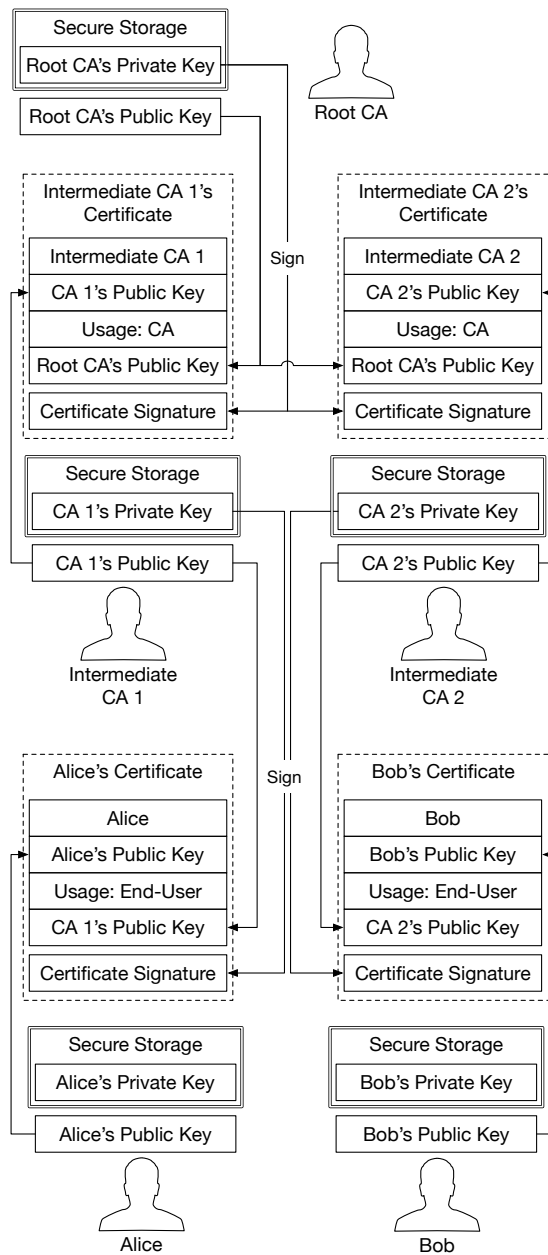
One of the main drawbacks of the CA system is that the CA's private key becomes a very attractive attack target. This issue is somewhat mitigated by minimizing the use of the CA's private key, which reduces the opportunities for its compromise. The authority described above becomes the *root CA*, and their private key is only used to produce certificates for the *intermediate CAs* who, in turn, are responsible for generating certificates for the other parties in the system, as shown in Figure 3.18.

In hierarchical CA systems, the only public key that gets distributed securely to all parties is the root CA's public key. Therefore, when two parties wish to interact, each party must present their own certificate, as well as the certificate of the issuing CA. For example, given the hierarchy in Figure 3.18, Alice would prove the authenticity of her public key to Bob by presenting her certificate, as well as the certificate of Intermediate CA 1. Bob would first use the steps in Figure 3.17 to validate Intermediate CA 1's certificate against the root CA's public key, which would assure him of the authenticity of Intermediate CA 1's public key. Bob would then validate Alice's certificate using Intermediate CA 1's public key, which he now trusts.

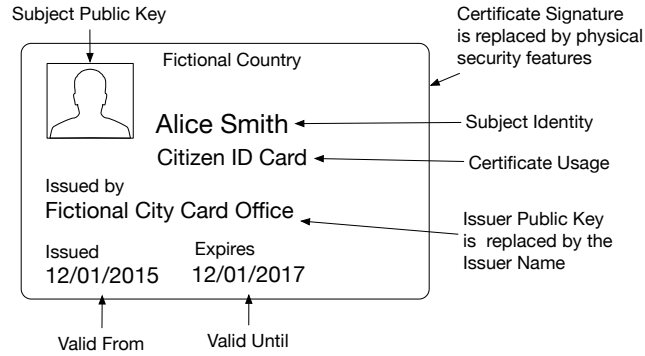
In most countries, the government issues ID cards for its citizens, and therefore acts as a certificate authority. An ID card, shown in Figure 3.19, is a certificate that binds a subject's identity, which is a full legal name, to the subject's physical appearance, which is used as a public key.

The CA system is very similar to the identity document (ID card) systems used to establish a person's identity, and a comparison between the two may help further the reader's understanding of the concepts in the CA system.

Each government's ID card issuing operations are regulated by laws, so an ID card's issue date can be used to track down the laws that make up its certification policy. The security of ID cards does not (yet) rely on cryptographic primitives. Instead, ID cards include physical security measures designed to deter tampering and prevent counterfeiting.



**Figure 3.18:** A hierarchical CA structure minimizes the usage of the root CA's private key, reducing the opportunities for it to get compromised. The root CA only signs the certificates of intermediate CAs, which sign the end users' certificates.



**Figure 3.19:** An ID card is a certificate that binds a subject’s full legal name (identity) to the subject’s physical appearance, which acts as a public key.

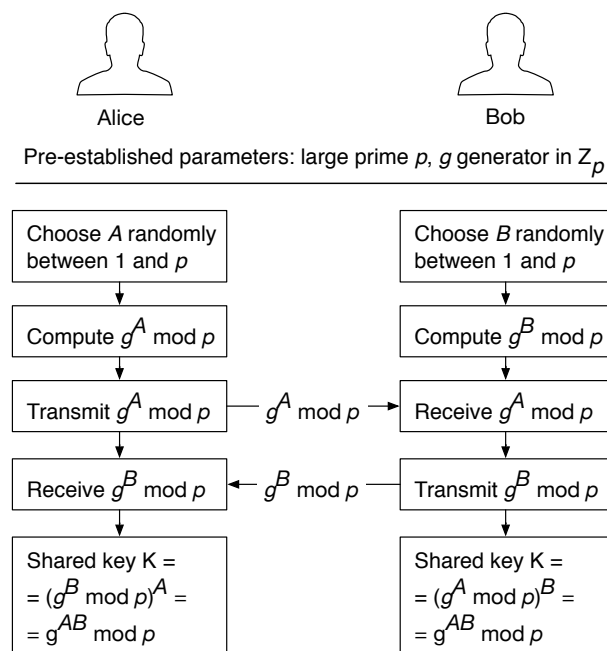
### 3.2.2 Key Agreement Protocols

The initial design of symmetric key primitives, introduced in § 3.1, assumed that when two parties wish to interact, one party generates a secret key and shares it with the other party using a communication channel with confidentiality and integrity guarantees. In practice, a pre-existing secure communication channel is rarely available.

*Key agreement protocols* are used by two parties to establish a shared secret key, and only require a communication channel with integrity guarantees. Figure 3.20 outlines the Diffie-Hellman Key Exchange (DKE) [Diffie and Hellman, 1976] protocol, which should give the reader an intuition for how key agreement protocols work.

This work is interested in using key agreement protocols to build larger systems, so we will neither explain the mathematical details in DKE, nor prove its correctness. We note that both Alice and Bob derive the same shared secret key,  $K = g^{AB} \bmod p$ , without ever transmitting  $K$ . Furthermore, the messages transmitted in DKE, namely  $g^A \bmod p$  and  $g^B \bmod p$ , are not sufficient for an eavesdropper Eve to determine  $K$ , because efficiently solving for  $x$  in  $g^x \bmod p$  is an open problem assumed to be very difficult.

Key agreement protocols require a communication channel with integrity guarantees. If an active adversary Eve can tamper with the



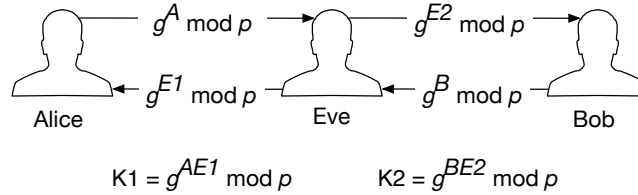
**Figure 3.20:** In the Diffie-Hellman Key Exchange (DKE) protocol, Alice and Bob agree on a shared secret key  $K = g^{AB} \bmod p$ . An adversary who observes  $g^A \bmod p$  and  $g^B \bmod p$  cannot compute  $K$ .

messages transmitted by Alice and Bob, she can perform a *man-in-the-middle* (MITM) attack, as illustrated in Figure 3.21.

In a MITM attack, Eve intercepts Alice's first key exchange message, and sends Bob her own message. Eve then intercepts Bob's response and replaces it with her own, which she sends to Alice. Eve effectively performs key exchanges with both Alice and Bob, establishing a shared secret with each of them, with neither Bob nor Alice being aware of her presence.

After establishing shared keys with both Alice and Bob, Eve can choose to observe the communication between Alice and Bob, by forwarding messages between them. For example, when Alice transmits a message, Eve can decrypt it using  $K_1$ , the shared key between herself and Alice. Eve can then encrypt the message with  $K_2$ , the key





**Figure 3.21:** Any key agreement protocol is vulnerable to a man-in-the-middle (MITM) attack. The active attacker performs key agreements and establishes shared secrets with both parties. The attacker can then forward messages between the victims, in order to observe their communication. The attacker can also send its own messages to either, impersonating the other victim.

established between Bob and herself. While Bob still receives Alice's message, Eve has been able to see its contents.

Furthermore, Eve can impersonate either party in the communication. For example, Eve can create a message, encrypt it with  $K2$ , and then send it to Bob. As Bob thinks that  $K2$  is a shared secret key established between himself and Alice, he will believe that Eve's message comes from Alice.

MITM attacks on key agreement protocols can be foiled by authenticating the party who sends the last message in the protocol (in our examples, Bob) and having them sign the key agreement messages. When a CA system is in place, Bob uses his public key to sign the messages in the key agreement and also sends Alice his certificate, along with the certificates for any intermediate CAs. Alice validates Bob's certificate, ensures that the subject identified by the certificate is whom she expects (Bob), and verifies that the key agreement messages exchanged between herself and Bob match the signature provided by Bob.

In conclusion, a key agreement protocol can be used to bootstrap symmetric key primitives from an asymmetric key signing scheme, where only one party needs to be able to sign messages.

### 3.3 Software Attestation Overview

The security of systems that employ trusted processors hinges on *software attestation*. The software running inside an *isolated container* es-

tablished by trusted hardware can ask the hardware to sign (§ 3.1.3) a small piece of *attestation data*, producing an *attestation signature*. Besides from the attestation data, the signed message includes a *measurement* that uniquely identifies the software inside the container. Therefore, an attestation signature can be used to convince a *verifier* that the attestation data was produced by a specific piece of software, which is hosted inside a container that is isolated by trusted hardware from outside interference.

Each hardware platform discussed in this section uses a slightly different software attestation scheme. Platforms differ by the amount of software that executes inside an isolated container, by the isolation guarantees provided to the software inside a container, and by the process used to obtain a container's measurement. The threat model and security properties of each trusted hardware platform follow directly from the design choices outlined above, so a good understanding of attestation is a prerequisite to discussing the differences between existing platforms.

### 3.3.1 Secure Remote Computation

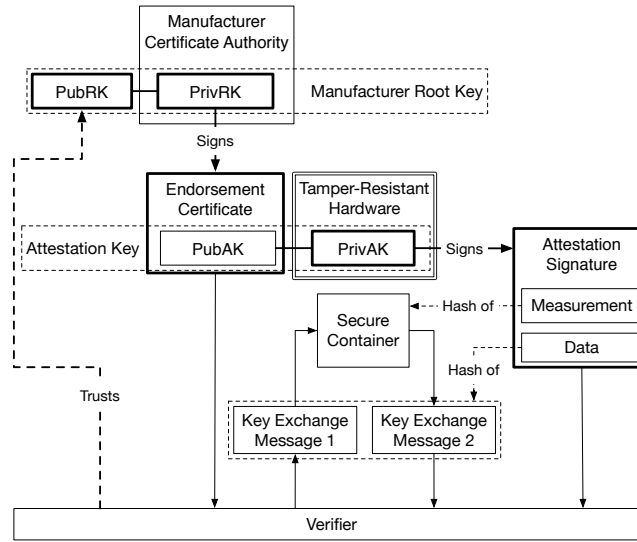
Secure remote computation (Figure 1.1) is the problem of executing software on a remote computer **owned and maintained by an untrusted party**, with some integrity and confidentiality guarantees. In the general setting, secure remote computation is an unsolved problem. Fully Homomorphic Encryption [Gentry, 2009] solves the problem for a limited family of computations, but has an impractical performance overhead [Naehrig et al., 2011].

### 3.3.2 Authenticated Key Agreement

Software attestation can be combined with a key agreement protocol (§ 3.2.2), as software attestation provides the authentication required by the key agreement protocol. The resulting protocol can assure a verifier that it has established a shared secret with a specific piece of software, hosted inside an isolated container created by trusted hardware. The next paragraph outlines the augmented protocol, us-

ing Diffie-Hellman Key Exchange (DKE) [Diffie and Hellman, 1976] as an example of the key exchange protocol.

The verifier starts executing the key exchange protocol, and sends the first message,  $g^A$ , to the software inside the secure container. The software inside the container produces the second key exchange message,  $g^B$ , and asks the trusted hardware to attest the cryptographic hash of both key exchange messages,  $h(g^A||g^B)$ . The verifier receives the second key exchange and attestation signature, and authenticates the software inside the secure container by checking all signatures along the *attestation chain* of trust shown in Figure 3.22.



**Figure 3.22:** The chain of trust in software attestation. The root of trust is a manufacturer key, which produces an endorsement certificate for the secure processor’s attestation key. The processor uses the attestation key to produce the attestation signature, which contains a cryptographic hash of the container and a message produced by the software inside the container.

The chain of trust used in software attestation is rooted at a signing key owned by the hardware manufacturer, which must be trusted by the verifier. The manufacturer acts as a Certificate Authority (CA, § 3.2.1), and provisions each secure processor that it produces with a unique *attestation key*, which is used to produce *attestation signature*.

*tures*. The manufacturer also issues an *endorsement certificate* for each secure processor's attestation key. The certificate indicates that the key is meant to be used for software attestation. The certification policy generally states that, at the very least, the private part of the attestation key be stored in tamper-resistant hardware, and only be used to produce attestation signatures.

A secure processor identifies each isolated container by storing a cryptographic hash of the code and data loaded inside the container. When the processor is asked to sign a piece of attestation data, it uses the cryptographic hash associated with the container as the measurement in the attestation signature. After a verifier validates the processor's attestation key using its endorsement certificate, the verifier ensures that the signature is valid, and that the measurement in the signature belongs to the software with which it expects to communicate. Having checked all links in the attestation chain, the verifier has authenticated the other party in the key exchange, and is assured that it now shares a secret with the software that it expects, running in an isolated container on hardware that it trusts.

### 3.3.3 The Role of Software Measurement

The measurement that identifies the software inside a secure container is always computed using a secure hashing algorithm (§ 3.1.3). Trusted hardware designs differ in their secure hash function choices, and in the data provided to the hash function. However, all designs share the principle that each step taken to build a secure container contributes data to its measurement hash.

The philosophy behind software attestation is that the computer's owner can load any software she wishes in a secure container. However, the computer owner is assumed to have an incentive to participate in a distributed system where the secure container she built is authenticated via software attestation. Without the requirement to undergo software attestation, the computer owner can build any container without constraints, which would make it impossible to reason about the security properties of the software inside the container.

By the argument above, a trusted hardware design based on software attestation must assume that each container is involved in software attestation, and that the remote party will refuse to interact with a container whose reported measurement does not match the expected value set by the distributed system's author.

For example, a cloud infrastructure provider should be able to use the secure containers provided by trusted hardware to run any software she wishes on her computers. However, the provider makes money by renting her infrastructure to customers. If security savvy customers are only willing to rent containers provided by trusted hardware, and use software attestation to authenticate the containers that they use, the cloud provider will have a strong financial incentive to build the customers' containers according to their specifications, so that the containers pass the software attestation.

A container's measurement is computed using a secure hashing algorithm, so the only method of building a container that matches an expected measurement is to follow the exact sequence of steps specified by the distributed system's author. The cryptographic properties of the secure hash function guarantee that if the computer's owner strays in any way from the prescribed sequence of steps, the measurement of the created container will not match the value expected by the distributed system's author, so the container will be rejected by the software attestation process.

Therefore, it makes sense to state that a trusted hardware design's measurement scheme guarantees that a property has a certain value in a secure container. The precise meaning of this phrase is that the property's value determines the data used to compute the container's measurement, so an expected measurement hash effectively specifies an expected value for the property. All containers in a distributed system that correctly uses software attestation will have the desired value for the given property.

For example, the measuring scheme used by trusted hardware designed for cloud infrastructure should guarantee that the container's memory was initialized using the customer's content, often referred to as an image.

### 3.4 Physical Attacks

Physical attacks are generally classified according to their cost, which factors in the equipment needed to carry out the attack and the attack's complexity. Joe Grand's DefCon presentation [Grand, 2004] provides a good overview with a large number of intuition-building figures and photos.

The simplest type of physical attack is a denial of service attack performed by disconnecting the victim computer's power supply or network cable. The threat models of most secure architectures exclude this attack from consideration, because denial of service can also be achieved by software attacks that compromise system software such as the hypervisor.

#### 3.4.1 I/O Port Attacks

Slightly more involved attacks rely on connecting a device to an existing port on the victim computer's case or motherboard (§ 2.9.1). A simple example is a *cold boot attack*, where the attacker plugs in a USB flash drive into the victim's case and causes the computer to boot from the flash drive loaded with malicious system software, which receives unrestricted access to the computer's peripherals.

More costly physical attacks that still require relatively little effort target the debug ports of various peripherals. The cost of these attacks is generally dominated by the expense of acquiring the development kits needed to connect to the debug ports. For example, recent Intel processors include the Generic Debug eXternal Connection (GDXC) [Yuffe et al., 2011, Kurts et al., 2011], which collects, filters, and exposes the data transferred by the uncore's ring network (§ 2.11.3), and reports it to an external debugger.

The threat models of secure architectures generally ignore debug port attacks, under the assumption that devices sold for general consumption have their debug ports irreversibly disabled. In practice, manufacturers have strong incentives to preserve debugging ports in production hardware, as this facilitates the diagnosis and repair of de-

fective units. Due to insufficient documentation on this topic, we do not survey GDXC-based attacks in this work.

### 3.4.2 Bus Tapping Attacks

More complex physical attacks consist of installing a device that taps a bus on the computer’s motherboard (§ 2.9.1). *Passive attacks* are limited to monitoring the bus traffic, whereas *active attacks* can modify the traffic, or even place new commands on the bus. *Replay attacks* are a notoriously challenging class of active attacks, where the attacker first records the bus traffic, and then selectively replays a subset of the traffic. Replay attacks bypass systems that rely on static signatures or HMACs, and generally aim to double-spend a limited resource.

The cost of bus tapping attacks is generally dominated by the cost of the equipment used to tap the bus, which increases with bus speed and complexity. For example, the flash module storing the computer’s firmware is connected to the PCH via an SPI bus (§ 2.9.1), which is simpler and much slower than the DDR bus connecting DRAM to the CPU. Consequently, tapping the SPI bus is much cheaper than tapping the DDR bus. For this reason, systems whose security relies on a cryptographic hash of the firmware will first copy the firmware into DRAM, hash the DRAM copy of the firmware, and then execute the firmware from DRAM.

Although the speed of the DDR DRAM link makes tapping very difficult, there are well-publicized records of successful attempts. The original Xbox console’s boot process was reverse-engineered via a passive tap on the DRAM bus [Huang, 2003], which showed that the firmware used to boot the console was partially stored in its south-bridge. The protection mechanisms of the PlayStation 3 hypervisor were subverted by an active tap on its memory bus [Hotz, 2010] that targeted the hypervisor’s page tables.

The Ascend secure processor (§ 4.10) demonstrated that concealing the DRAM addresses accessed by a program is orders of magnitude more expensive than protecting the data in memory. Therefore, we are chiefly interested in analyzing attacks that tap the DRAM bus and collect information on the address lines. These attacks use the

same equipment as normal DRAM bus tapping attacks, but require a significantly more involved analysis to learn useful information from the gathered data. One of the difficulties of such attacks stems from the fact that the memory addresses observed on the DRAM bus are generally very different from the application's memory access patterns due to the behavior of cache hierarchies and multiprogramming in modern processors (§ 2.11). At the time of this writing, we are not aware of any successful attack based on tapping the address lines of a DRAM bus and analyzing the sequence of memory addresses.

### **3.4.3 Attacks on the Processor Package or Die**

The most equipment-intensive physical attacks involve removing a chip's packaging and directly interacting with its electrical circuits. These attacks generally take advantage of equipment and techniques that were originally developed to diagnose design and manufacturing defects in integrated circuits. [Beck, 1998] covers these techniques in depth.

The cost of chip attacks is dominated by the required equipment, although the reverse-engineering involved is also non-trivial. This cost grows very rapidly as the circuit components shrink with advances in fabrication technology. At the time of this writing, the latest widely available state-of-the-art processor systems have a 14nm feature size, which requires ion beam microscopy for such analysis.

The least expensive classes of chip attacks are destructive, and only require imaging the chip's circuitry. These attacks rely on a microscope capable of capturing the necessary details in each layer, and equipment for mechanically removing each layer and exposing the layer below it to the microscope.

Imaging attacks generally target global secrets shared by all devices in a family, such as ROM masks that store global encryption keys or secret boot code. They are also used to reverse-engineer undocumented functionality, such as debugging backdoors. E-fuses and polyfuses are particularly vulnerable to imaging attacks, because of their relatively large sizes.



Non-destructive passive chip attacks include measuring the voltages across a module at specific times while the chip is active. These attacks are orders of magnitude more expensive than destructive imaging attacks because the attacker must take care to maintain the integrity of the chip's circuitry, and therefore cannot de-layer the chip and has limited visibility and access.

The simplest active attacks on a chip create or destroy an electric connection between two components. For example, the debugging functionality in many devices is disabled by “blowing” an e-fuse. Once this e-fuse is located, an attacker can reconnect its two ends, effectively undoing the “blowing” operation. More expensive attacks involve changing voltages across a component as the chip is operating, and are typically used to reverse-engineer complex circuits.

Surprisingly, active attacks are not significantly more expensive to carry out than passive non-destructive attacks. This is because the tools used to measure the voltage across specific components are not very different from the tools that can tamper with the chip's electric circuits. Therefore, once an attacker develops a process for accessing a module without destroying the chip's circuitry, the attacker can use the same process for both passive and active attacks.

At the architectural level, we cannot address physical attacks against the CPU's chip package. Active attacks on the CPU change the computer's execution semantics, leaving us without any hardware that can be trusted to make security decisions. Passive attacks can read the private data that the CPU is processing. Therefore, many secure computing architectures assume that the processor chip package is invulnerable to physical attacks.

Thankfully, physical attacks can be deterred by reducing the value that an attacker obtains by compromising an individual chip. As long as this value is below the cost of carrying out the physical attack, a system's designer can hope that the processor's chip package will not be targeted by the physical attacks.

Architects can reduce the value of compromising an individual system by avoiding shared secrets, such as global encryption keys. Chip designers can increase the cost of a physical attack by not stor-

ing a platform's secrets in hardware that is vulnerable to destructive attacks, such as e-fuses.

### **3.4.4 Power Analysis Attacks**

An entirely different approach to physical attacks consists of indirectly measuring the power consumption of a computer system or its components. The attacker takes advantage of a known correlation between power consumption and the computed data, and learns some property of the data from the observed power consumption.

The earliest power analysis attacks have directly measured the processor chip's power consumption. For example, [Kocher et al., 1999] describes a simple power analysis (SPA) attack that exploits the correlation between the power consumed by a smart card chip's CPU and the type of instruction it executed, and learned a DSA key that the smart card was supposed to safeguard.

While direct power analysis attacks necessitate some equipment, their costs are dominated by the complexity of the analysis required to learn the desired information from the observed power trace which, in turn, is determined by the complexity of the processor's circuitry. Today's smart cards contain special circuitry [Tiri et al., 2002] and use hardened algorithms [Herbst et al., 2006] designed to frustrate power analysis attacks.

Recent work demonstrated successful power analysis attacks against a system as complex as a full-blown out-of-order Intel processor using inexpensive off-the-shelf sensor equipment. [Genkin et al., 2013] extracts an RSA key from GnuPG running on a laptop using a microphone that measures its acoustic emissions. [Genkin et al., 2014] and [Genkin et al., 2015] extract RSA keys from power analysis-resistant implementations using a voltage meter and a radio. All of these attacks can be performed quite easily by a disgruntled data center employee.

Unfortunately, power analysis attacks can be extended to displays and human input devices, which cannot be secured in any reasonable manner. For example, [Van Eck, 1985] documented a very early attack that measures the radiation emitted by a CRT display's ion beam to reconstitute the image on a computer screen in a different

room. [Kuhn, 2005] extended the attack to modern LCD displays. [Zhuang et al., 2009] used a directional microphone to measure the sound emitted by a keyboard and learn the password that its operator typed. [Owusu et al., 2012] applied similar techniques to learn a user's input on a smartphone's on-screen keyboard, based on data from the device's accelerometer.

In general, power attacks cannot be addressed at the architectural level, as they rely on implementation details that are decided during the manufacturing process. Therefore, it is unsurprising that the secure computing architectures described in § 4 do not protect against power analysis attacks.

### 3.5 Privileged Software Attacks

The rest of this section points to successful exploits that execute at each of the privilege levels described in § 2.3, motivating the SGX design decision to assume that all privileged software on the computer is malicious. [Rutkowska, 2015] describes all programmable hardware inside Intel computers, and outlines the security implications of compromising the software running it.

SMM, the most privileged execution level, is only used to handle a specific kind of interrupts (§ 2.12), namely *System Management Interrupts* (SMI). SMIs were initially designed exclusively for hardware use, and were only triggered by asserting a dedicated pin (SMI#) in the CPU's chip package. However, in modern systems, system software can generate an SMI by using the LAPIC's IPI mechanism. This opens up the avenue for SMM-based software exploits.

The SMM handler is stored in *System Management RAM* (SMRAM) which, in theory, is not accessible when the processor isn't running in SMM. However, its protection mechanisms were bypassed multiple times [Duflot et al., 2006, Rutkowska and Wojtczuk, 2008, Wojtczuk and Rutkowska, 2009a, Kallenberg et al., 2014], and SMM-based rootkits [Wecherowski, 2009, Embleton et al., 2010] have been demonstrated. Compromising the SMM grants an attacker access to all software on the computer, as SMM is the most privileged execution mode.

Xen [Zhang and Dong, 2008] is a very popular representative of the family of hypervisors that run in VMX root mode and use hardware virtualization. At 150,000 lines of code [xen, 2015], Xen’s codebase is relatively small, especially when compared to a kernel. However, Xen still has had over 40 security vulnerabilities patched in **each** of the last three years (2012-2014) [cve, 2014b].

[McCune et al., 2010] proposes using a very small hypervisor together with Intel TXT’s dynamic root of trust for measurement (DRTM) to implement trusted execution. [Vasudevan et al., 2010] argues that a dynamic root of trust mechanism, like Intel TXT, is necessary to ensure a hypervisor’s integrity. Unfortunately, the TXT design requires an implementation complex enough that exploitable security vulnerabilities have crept in [Wojtczuk et al., 2009, Wojtczuk and Rutkowska, 2011]. Furthermore, any SMM attack can be used to compromise TXT [Wojtczuk and Rutkowska, 2009b].

The monolithic kernel design leads to many opportunities for security vulnerabilities in kernel code. Linux is by far the most popular kernel for IaaS cloud environments. Linux has *17 million* lines of code [Anthony, 2014], and has had over 100 security vulnerabilities patched in **each** of the last three years (2012-2014) [cve, 2014a, Chen et al., 2011].

### 3.6 Software Attacks on Peripherals

Threat models for secure architectures generally only consider software attacks that directly target other components in the software stack running on the CPU. This assumption results in security arguments with the very desirable property of not depending on implementation details, such as the structure of the motherboard hosting the processor package.

The threat models mentioned above must classify attacks mounted via motherboard components other than the CPU as physical attacks. Unfortunately, these models miscategorize the attacks described in this section, which can be carried out solely by executing software on the victim processor. The incorrect classification matters a great deal in cloud computing scenarios, where physical attacks often deemed out of scope as prohibitively expensive to carry out.

By way of specific example, this section discusses attacks primarily in the context of Intel’s Core processors, which at the time of publication are by far the most widely available processor systems in their class, and offer a well-studied target for type of attack.

### 3.6.1 PCI Express Attacks

The PCIe bus (§ 2.9.1) allows any device connected to the bus to perform *Direct Memory Access* (DMA), reading from and writing to blocks of addresses in the computer’s DRAM without the involvement of a CPU core. Each device is assigned a range of DRAM addresses via a standard PCI configuration mechanism, but can perform DMA on DRAM addresses outside of that range.

Without any additional protection mechanisms, an attacker who compromises system software can take advantage of programmable devices to access any DRAM region, yielding capabilities that were traditionally associated with a DRAM bus tap. For example, an early implementation of Intel TXT [Grawrock, 2009] was compromised by programming a PCIe network interface card (NIC) to read TXT-reserved DRAM via DMA transfers [Wojtczuk and Rutkowska, 2011]. Recent versions have addressed this attack by adding extra security checks in the DMA bus arbiter. § 4.5 provides a more detailed description of Intel’s TXT.

### 3.6.2 DRAM Attacks

The rowhammer DRAM bit-flipping attack [Kim et al., 2014, Seaborn and Dullien, 2015, Gruss et al., 2015] is an example of a different class of software attacks that exploit design defects in the computer’s hardware. Rowhammer took advantage of the fact that some mobile DRAM devices (§ 2.9.1) refreshed the DRAM’s contents slowly enough that repeatedly changing the contents of a memory cell could impact the charge stored in a neighboring cell, which resulted in changing the bit value obtained from reading the cell. By carefully targeting specific memory addresses, the attackers caused bit flips in the page tables used by the CPU’s address translation (§ 2.5) mechanism, and in other data structures used to make security decisions.

The defect exploited by the rowhammer attack most likely stems from an unfortunate and incorrect design assumption. The DRAM engineers likely only considered non-malicious software and assumed that an individual DRAM cell is not often accessed, as repeated accesses to the same memory address would be absorbed by the CPU's caches (§ 2.11). However, malicious software can take advantage of the **CLFLUSH** instruction, which flushes the cache line that contains a given DRAM address. **CLFLUSH** is intended as a method for applications to extract more performance out of the cache hierarchy, and is therefore available to software running at all privilege levels. Rowhammer exploited the combination of **CLFLUSH**'s availability and the DRAM engineers' incorrect assumptions, to obtain capabilities that are normally associated with an active DRAM bus attack.

### 3.6.3 The Performance Monitoring Side Channel

Intel's *Software Development Manual* (SDM) [Int, 2015g] and *Optimization Reference Manual* [Int, 2014c] describe a vast array of performance monitoring events exposed by recent Intel processors, such as branch mispredictions (§ 2.10). The SDM also describes digital temperature sensors embedded in each CPU core, whose readings are exposed using Model-Specific Registers (MSRs) (§ 2.4) that can be read by system software.

An attacker able to compromise a computer's system software and gain access to the performance monitoring events or the temperature sensors can obtain the information needed to carry out a power analysis attack, which normally requires physical access to the victim computer and specialized equipment. Simpler yet, the attacker may learn private information from various performance counters affected by the victim's execution.

### 3.6.4 Attacks on the Boot Firmware and Intel ME

Virtually all motherboards store the firmware used to boot the computer in a flash memory module (§ 2.9.1) that can be written by system software. This implementation strategy provides an inexpensive avenue for deploying firmware bug fixes. On the other hand, an attack

that compromises the system software can subvert the firmware update mechanism to inject malicious code into the platform firmware. The malicious code can be used to carry out a cold boot attack, which is typically considered a physical attack. Furthermore, malicious firmware can execute code at the highest software privilege level, System Management Mode (SMM, § 2.3). Last, malicious firmware can modify the system software as it is loaded during the boot process. These avenues give the attacker many capabilities that have traditionally been associated with DRAM bus tapping attacks.

The Intel Management Engine (ME) [Ruan, 2014] loads its firmware from the same flash memory module as the main computer, which opens up the possibility of compromising its firmware. Due to the vast management capabilities (§ 2.9.2) of a ME, if compromised, it would offer an attacker similar capabilities to active probes on the DRAM bus, the PCI bus, and the System Management bus (SMBus), as well as a wealth of power meters. Thanks to its direct access to the motherboard's Ethernet PHY, the probe would be able to communicate with the attacker while the computer is in the Soft-Off state, also known as S5, where the computer is mostly powered off, but is still connected to a power source. The ME has significantly less computational power than probe equipment, however, as it uses low-power embedded components, such as a 200-400MHz execution core, and about 600KB of internal RAM.

The computer and ME firmware are protected by a few security measures. The first line of defense is a security check in the firmware's update service, which only accepts firmware updates that have been digitally signed by a manufacturer key that is hard-coded in the firmware. This protection can be circumvented with relative ease by foregoing the firmware's update services, and instead accessing the flash memory chip directly, via the PCH's SPI bus controller.

The deeper, more powerful, lines of defense against firmware attacks are rooted in the CPU and ME's hardware. The bootloader in the ME's ROM will only load flash firmware that contains a correct signature generated by a specific Intel RSA key. The ME's boot ROM contains the SHA-256 cryptographic hash of the RSA public key, and uses it to validate the full Intel public key stored in the signature. Similarly, the

microcode bootstrap process in recent CPUs will only execute firmware in an Authenticated Code Module (ACM, § 2.13.2) signed by an Intel key whose SHA-256 hash is hard-coded in the microcode ROM.

However, both the computer firmware security checks [Wojtczuk and Tereshkin, 2010, Furtak et al., 2014] and the ME security checks [Tereshkin and Wojtczuk, 2009] have been subverted in the past. While the approaches described above are theoretically sound, the intricate details and complex interactions in Intel-based systems make it very likely that security vulnerabilities creep into implementations. Further proving this point, a security analysis [Ververis, 2010] found that early versions of Intel’s Active Management Technology (AMT), the flagship ME application, contained assorted of security issues that allowed an attacker to completely take over a computer whose ME firmware contained the AMT application.

### **3.6.5 Software Attacks on Peripheral Devices**

The attacks described in this section show that a system whose threat model assumes no physical access attacks must be designed with an understanding of all of the system’s buses, and the programmable devices that may be attached to them. The system’s security analysis must argue that the devices cannot be used in physical-like attacks. The argument will rely on barriers that prevent untrusted software running on the CPU from communicating with other programmable devices, and on barriers that prevent compromised programmable devices from tampering with sensitive buses or DRAM.

Unfortunately, at the time of publication, the ME, PCH and DMI are proprietary in Intel’s processors and remain largely undocumented. We therefore cannot assess the security of the measures set in place to protect the ME from compromise, and we cannot rigorously reason about the impact of a compromised ME on the security of a computer system.



### 3.7 Address Translation Attacks

§ 3.5 argues that today's system software is all but guaranteed to have security vulnerabilities. This suggests that a cautious secure architecture should avoid including the system software in the TCB.

However, removing the system software from the TCB requires the architecture to provide a method for isolating sensitive application code from the untrusted system software. This is typically accomplished by designing a mechanism for loading application code into isolated containers whose contents can be certified via software attestation (§ 3.3). One of the more difficult problems these designs face is that application software relies on the memory management services provided by the system software, which is now untrusted.

For example, Intel's SGX [McKeen et al., 2013, Anati et al., 2013], leaves the system software in charge of setting up the page tables (§ 2.5) used by address translation, inspired by Bastion [Champagne and Lee, 2010], but instantiates access checks that prevent the system software from directly accessing the isolated container's memory.

This section discusses some attacks that become relevant when the application software does not trust the system software, which is in charge of the page tables. Understanding these attacks is a prerequisite to reasoning about the security properties of architectures with this threat model. For example, many of the mechanisms in Intel's SGX seek to prevent a subset of the attacks described here.

#### 3.7.1 Passive Attacks

System software uses the CPU's address translation feature (§ 2.5) to implement page swapping, where infrequently used memory pages are evicted from DRAM to a slower storage medium. Page swapping relies the accessed (A) and dirty (D) page table entry attributes (§ 2.5.3) to identify the DRAM pages to be evicted, and on a page fault handler (§ 2.8.2) to bring evicted pages back into DRAM when they are accessed.

Unfortunately, the features that support efficient page swapping turn into a security liability when the system software managing the

page tables is not trusted by the application software using the page tables. The system software can be blocked from reading the application's memory directly by placing the application in an isolated container. However, potentially malicious system software may infer partial information about the application's memory access patterns, by observing the application's page faults and page table attributes.

We consider this class of attacks to be passive attacks that exploit the CPU's address translation feature. It may seem that the page-level memory access patterns provided by these attacks are not very useful. However, [Xu et al., 2015] describes how this attack can be carried out against Intel's SGX, and implements the attack in a few practical settings. In one scenario, which is particularly concerning for medical image processing, the outline of a JPEG image is inferred while the image is decompressed inside a container protected by SGX's isolation guarantees.

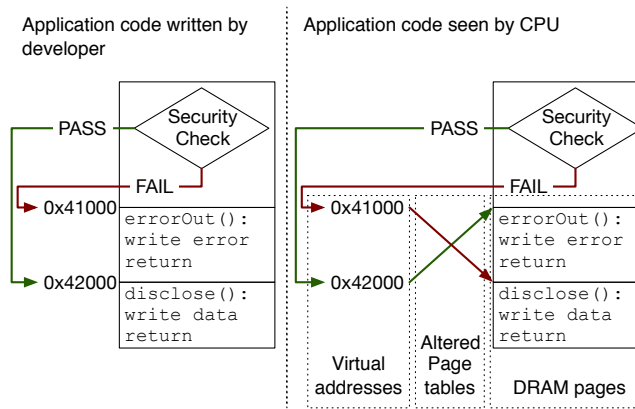
### 3.7.2 Straightforward Active Attacks via Address Translation

We define active address translation attacks to be the class of attacks where malicious system software modifies the page tables used by an application in a way that breaks the virtual memory abstraction (§ 2.5). Memory mapping attacks do not include scenarios where the system software breaks the memory abstraction by directly writing to the application's memory pages.

We begin with an example of a straightforward active attack. In this example, the application inside a protected container performs a security check to decide whether to disclose some sensitive information. Depending on the security check's outcome, the enclave code either calls a `errorOut` procedure, or a `disclose` procedure. The simplest version of the attack assumes that each procedure's code starts at a page boundary, and takes up less than a page. These assumptions are relaxed in more complex versions of the attack.

In the most straightforward setting, the malicious system software directly modifies the page tables of the application inside the container, as shown in Figure 3.23, so the virtual address intended to store the `errorOut` procedure is actually mapped to a DRAM page

that contains the `disclose` procedure. Without any security measures in place, when the application's code jumps to the virtual address of the `errorOut` procedure, the CPU will execute the code of the `disclose` procedure instead.

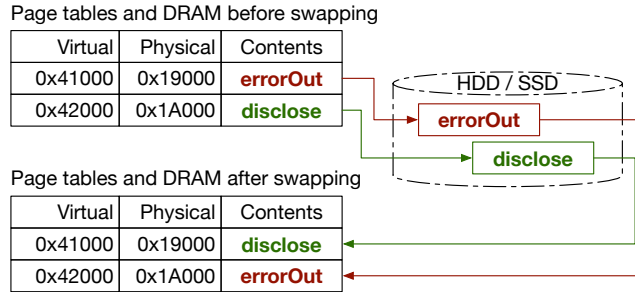


**Figure 3.23:** An example of an active memory mapping attack. The application's author intends to perform a security check, and only call the procedure that discloses the sensitive information if the check passes. Malicious system software maps the virtual address of the procedure that is called when the check fails, to a DRAM page that contains the disclosing procedure.

### 3.7.3 Active Attacks Using Page Swapping

The most obvious active attacks on virtual memory can be defeated by a naive address check. By verifying the virtual address of each DRAM page belonging to a protected container, the system would ensure integrity of address mappings for sensitive pages. This protection mechanism is, however, defeated by a more subtle active attack exploiting architectural support for page swapping. Figure 3.24 illustrates an attack that does not modify the application's page tables, but produces the same corrupted CPU view of the application as the straightforward attack described above.

In this attack, malicious system software evicts the pages that contain the `errorOut` and `disclose` procedures from DRAM to a slower medium, such as a hard disk. The system software exchanges



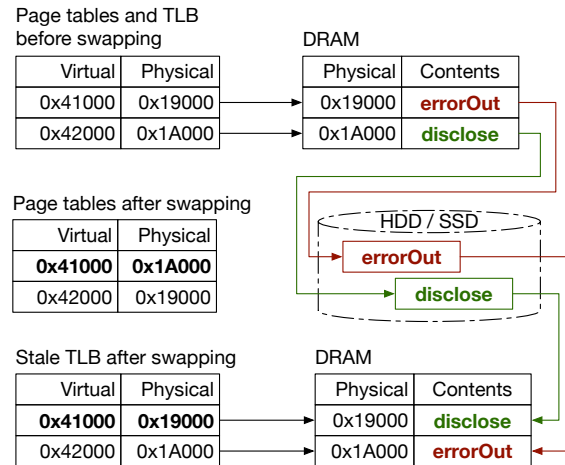
**Figure 3.24:** An active memory mapping attack where the system software does not modify the page tables. Instead, two pages are evicted from DRAM to a slower storage medium. The malicious system software swaps the two pages’ contents then brings them back into DRAM, building the same incorrect page mapping as the direct attack shown in Figure 3.23. This attack defeats protection measures that rely on tracking the virtual and disk addresses for DRAM pages.

the hard disk bytes storing the two pages, and then brings the two pages back into DRAM. Remarkably, all of the steps performed by this attack are indistinguishable from legitimate page swapping activity, with the exception of the I/O operations that exchange the disk bytes storing evicted pages.

The subtle attack described in this section can be defeated by cryptographically binding the contents of each page that is evicted from DRAM to the virtual address to which the page should be mapped. The cryptographic primitive (§ 3.1) used to perform the binding must obviously guarantee integrity by detecting an attack that alters the data of a page. Furthermore, it must also guarantee freshness, in order to foil replay attacks where the system software “undoes” an application’s writes by evicting one of its DRAM pages to disk and bringing in a prior version of the same page.

### 3.7.4 Active Attacks Based on TLBs

Today’s multi-core architectures can be subjected to an even more subtle active attack, illustrated in Figure 3.25, which can bypass any protection measures that solely focus on the integrity of the page tables.



**Figure 3.25:** An active memory mapping attack where the system software does not invalidate a core’s TLBs when it evicts two pages from DRAM and exchanges their locations when reading them back in. The page tables are updated correctly, but the core with stale TLB entries has the same incorrect view of the protected container’s code as in Figure 3.23.

For reasons of performance, each execution core caches address translations in the core’s translation look-aside buffer (TLB, § 2.11.5). To reduce complexity, the TLBs are not maintained by the cache coherence protocol, and must be managed by system software in order to remain consistent with the system’s access control policies. Specifically, the system software is responsible for invalidating TLB entries across *all* cores whenever it modifies the page tables.

Malicious system software can exploit the design decisions above by carrying out the following attack. While the same software used in the previous examples is executing on core 0, system software executes on core 1 and evicts the **errorOut** and **disclose** pages from DRAM. As in the previous attack, the system software loads the **disclose** code in the DRAM page that previously held **errorOut**. In this attack, however, the system software also updates the page tables.

Core 1, where the system software executed, has a view of the code as intended by the application developer, meaning the attack will undergo any security checks that rely upon cryptographic associations

between page contents and page table data, as long as the checks are performed by the core used to load pages back into DRAM. However, core 0, which executes the protected container's code, uses memory mappings from obsolete page tables, as the system did not invalidate its TLB entries. Assuming the TLBs are not subjected to any additional security checks, this attack causes the same private information leak as in previous examples.

In order to avoid the attack described in this section, the trusted software or hardware that implements protected containers must also ensure that the system software invalidates the relevant TLB entries on all cores when it evicts a page from a protected container to DRAM.

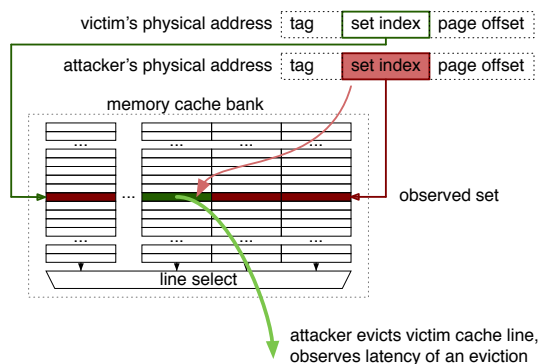
### **3.8 Cache Timing Attacks**

Cache timing attacks [Banescu, 2011] are a powerful class of software attacks that can be mounted entirely by an unprivileged attacker (ring 3, § 2.3). Cache timing attacks do not reveal information by directly reading the victim's memory, but by indirectly observing the victim's memory access pattern via their use of the system's caches. These attacks therefore sidestep address translation-based isolation measures (§ 2.5) implemented in modern kernels and hypervisors.

#### **3.8.1 Theory**

Cache timing attacks exploit the unfortunate dependency between the part of a computer's memory subsystem hosting the freshest copy of a chunk of memory, and the latency of the corresponding access. A cache miss requires at the minimum a lookup in the core's L1 cache, and accesses to subsequent caches in the memory hierarchy if the address is not present in the L1. If the cache is full and dirty eviction must occur, further latency is incurred due to a write-back of evicted data. On the Intel architecture, the latency between a cache hit and a miss can be easily resolved via the RDTSC and RDTSCP instructions (§ 2.4), which expose a high-frequency cycle counter. These instructions have been designed for benchmarking and optimizing software, and provide a high-resolution measure of time to unprivileged (ring 3) software.

The fundamental tool of a cache timing attack is the attacker's facility to measure the latency of their own memory accesses, as affected by the victim's use of the cache. A large multitude of addresses compete for any given cache set, giving the attacker ample room to arrange this interference, and observe the victim's use of the contested cache sets by monitoring the latency of the attacker's memory operations. The memory locations are chosen so that they map to the same cache lines as those of some interesting memory locations in a victim process, in a cache that is shared between the attacker and the victim. This family of attacks (as exemplified in Figure 3.26) generally requires the attacker to know cache sizes, organization, and eviction behavior (§ 2.11.2), all of which are readily available.



**Figure 3.26:** A cache timing attack via shared cache sets corresponding to disjoint physical addresses. The attacker measures the availability of their own cache sets to indirectly observe the victim's use of specific cache sets and therefore the victim's memory access pattern.

Armed with this knowledge, the attacker process begins with a series of operations that forces evictions on all cache sets corresponding to an address of interest in the victim's memory. The exact mechanism varies by attack. A straightforward eviction via dedicated instructions is available to the attacker on shared pages (such as shared library code, or pages de-duplicated by system software for efficiency). The attacker can also exploit their knowledge of the cache eviction behavior by performing a series of memory accesses on their own virtual address

space in a way that fills all cache sets competing with the victim's address of interest, causing these to be evicted.

This forces the victim's cache lines out of the cache and into DRAM (or lower levels of the cache hierarchy). When the victim process is scheduled and executes, any accesses to the monitored addresses must bring the corresponding lines back into the cache.

The attacker periodically repeats their forced eviction of the victim's lines, and measures the victim's use of the cache since last eviction. This is accomplished in one of several ways, again depending on the attack. In case of shared physical pages, the attacker is able to measure the latency of a direct read to the address of interest. A high latency indicates the victim has not accessed the address of interest, while a low latency indicates the line was re-introduced into the cache by the victim's execution. In other cases, the attacker must infer the victim's use of the cache by monitoring the latency of accesses to the attacker's own competing cache sets. By monitoring the time needed to re-fill all relevant cache sets with the attacker's lines, they can and detect evictions caused by the victim's execution. In some cases, the attacker can further resolve victim stores from loads, as evictions of dirty cache lines are slower than clean ones.

Over time, the attacker collects evidence of the victim's execution and learns partial information of the victim's memory access pattern. If the victim processes sensitive information using data-dependent control flow or data access pattern, the attacker may be able to infer this information from the observed memory access pattern.

### **3.8.2 Practical Considerations**

Cache timing attacks require control over a software process that shares cache sets with the victim process in any of the system's cache hierarchy. A cache timing attack that targets the L2 cache relies on the system software to co-locate the attacker thread with the victim thread on the same physical core, whereas an attack on the L3 (last level) cache can be performed by any logical processor on the same CPU. The latter attack relies on the fact that the L3 cache is inclusive, which greatly simplifies the processor's cache coherence implementation (§ 2.11.3).



The cache sharing requirement implies that L3 cache attacks are feasible in an IaaS environment, whereas L1 and L2 cache attacks are a significant concern when untrusted software runs at any privilege level alongside a sensitive process managed by the same operating system.

Out-of-order execution (§ 2.10) can introduce noise in cache timing attacks. First, memory accesses may not be performed in program order, which can impact the lines selected by the cache eviction algorithms. Second, out-of-order execution may result in cache fills that do not correspond to executed instructions. For example, a load that follows a faulting instruction may be scheduled and executed before the fault is detected.

Cache timing attacks must account for speculative execution, as mispredicted memory accesses may cause cache fills, causing the attacker to observe cache fills that do not correspond to instructions executed by the victim software. Memory prefetching adds further noise in form of cache fills that are informed by but are not the result of instructions in the victim code.

### 3.8.3 Known Cache Timing Attacks

Despite these difficulties, cache timing attacks are known to retrieve cryptographic keys used by numerous cryptosystems, including at the time of this writing AES [Osvik et al., 2006, Bonneau and Mironov, 2006], RSA [Brumley and Boneh, 2005], Diffie-Hellman [Kocher, 1996], and elliptic-curve cryptography [Brumley and Tuveri, 2011].

Early attacks required access to the victim’s CPU core, but more sophisticated recent attacks [Yarom and Falkner, 2013, Liu et al., 2015] are able to use the L3 (last-level) cache, which is shared by all cores on a CPU die. L3-based attacks can be particularly devastating in cloud computing scenarios, where running software on the same computer as a victim application only requires modest statistical analysis and a small payment [Ristenpart et al., 2009]. Another recently demonstrated class of cache timing attacks uses JavaScript code loaded as part of a web page visited by a Web browser [Oren et al., 2015], meaning these attacks are extremely easy to deploy.

Given this pattern of vulnerabilities, ignoring cache timing attacks is dangerously similar to ignoring the string of demonstrated attacks which led to the deprecation of SHA-1 [nis, 2014, goo, 2014, mic, 2016].

### **3.8.4 Defending against Cache Timing Attacks**

Fortunately, invalidating any of the preconditions for cache timing attacks is sufficient for defending against them. The easiest precondition to focus on is that the attacker must have access to memory locations that map to the same sets in a cache as the victim's memory. This assumption can be invalidated by the judicious use of a cache partitioning scheme.

Performance concerns aside, the main difficulty associated with cache partitioning schemes is that they must be implemented by a trusted party. When the system software is trusted, it can (for example) use the principles behind page coloring [Taylor et al., 1990, Kessler and Hill, 1992] to partition the caches [Lin et al., 2008] between mutually distrusting parties. This comes down to setting up the page tables in such a way that no two mutually distrusting software modules are stored in physical pages that map to the same sets in any cache. However, if the system software cannot be trusted, the cache partitioning scheme must be implemented by hardware.

The other interesting precondition is that the victim must access its memory in a data-dependent fashion that allows the attacker to infer private information from the observed memory access pattern. It becomes tempting to think that cache timing attacks can be prevented by eliminating data-dependent memory accesses from all code handling sensitive data.

However, removing data-dependent memory accesses is difficult to accomplish in practice because instruction fetches must also be taken into consideration. [Käsper and Schwabe, 2009] gives an idea of the level of effort required to remove data-dependent accesses from AES, which is a relatively simple data processing algorithm. At the time of this writing, we are not aware of any approach that scales to large pieces of software.

While the focus of this section is on cache timing attacks, we must emphasize that any sharing of resources among mutually distrusting entities may leak private information via the availability of the shared resource over time. One worrying example is hyper-threading (§ 2.9.4), where each CPU core implements two logical processors, and the threads executing on these two logical processors share execution units. An attacker able to run a process on a logical processor co-located on a core with a victim process can use `RDTSCP` [Petters and Farber, 1999] to learn which execution units are in use, and infer information about the instructions executed by the victim process.

# 4

---

## A Survey of Secure Processors

---

This section describes the broad landscape of trusted hardware projects in cursory terms. Table 4.1 summarizes the security properties of SGX and the other trusted hardware presented here.

### 4.1 The IBM 4765 Secure Coprocessor

Secure coprocessors [Yee, 1994] encapsulate an entire computer system, including a CPU, a cryptographic accelerator, caches, DRAM, and an I/O controller within a tamper-resistant environment. The enclosure includes hardware that deters attacks, such as a Faraday cage, as well as an array of sensors that can detect tampering attempts. The secure coprocessor destroys the secrets that it stores when an attack is detected. This approach has good security properties against physical attacks, but tamper-resistant enclosures are very expensive [Anderson, 2001], relative to the cost of a computer system.

The IBM 4758 [Smith and Weingart, 1999], and its most current-day successor, the IBM 4765 [nis, 2012] (shown in Figure 4.1) are representative examples of secure coprocessors. The 4758 was certified to

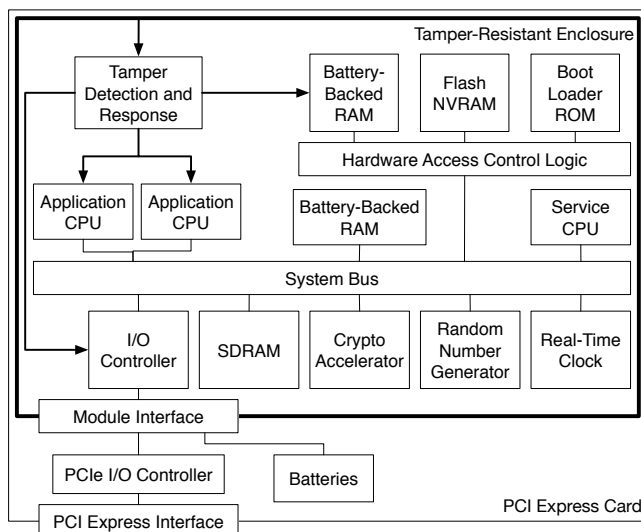
Table 4.1: Security features overview for the trusted hardware projects related to Intel's SGX.

| Attack                                | TrustZone                                    | TPM  | TPM+TXT                                    | SGX   | XOM  |
|---------------------------------------|--|--|--|---|--|
| Malicious containers (direct probing) | N/A (secure world is trusted)                | N/A (The computer is one container)        | N/A (Does not allow concurrent containers) | Access checks on TLB misses                         | Identifier tag checks                      |
| Malicious OS (direct probing)         | Access checks on TLB misses                  | N/A (OS measured and trusted)              | Host OS preempted during late launch       | Access checks on TLB misses                         | OS has its own identifier                  |
| Malicious hypervisor (direct probing) | Access checks on TLB misses                  | N/A (Hypervisor measured and trusted)      | Hypervisor preempted during late launch    | Access checks on TLB misses                         | N/A (No hypervisor support)                |
| Malicious firmware                    | N/A (firmware is a part of the secure world) | CPU microcode measures PEI firmware        | SINIT ACM signed by Intel key and measured | SMM handler is subject to TLB access checks         | N/A (Firmware is not active after booting) |
| Malicious containers (cache timing)   | N/A (secure world is trusted)                | N/A (Does not allow concurrent containers) | N/A (Does not allow concurrent containers) | ×   | ×  |
| Malicious OS (page fault recording)   | Secure world has own page tables             | N/A (OS measured and trusted)              | Host OS preempted during late launch       | ×   | N/A (Paging not supported)                 |
| Malicious OS (cache timing)           | ×  | N/A (OS measured and trusted)              | Host OS preempted during late launch       | ×   | ×  |
| DMA from malicious peripheral         | On-chip bus bounces secure world accesses    | ×  | IOMMU bounces DMA into TXT memory range    | IOMMU bounces DMA into PRM                          | Equivalent to physical DRAM access         |
| Physical DRAM read                    | Secure world limited to on-chip SRAM         | ×  | ×  | Undocumented memory encryption engine               | DRAM encryption                            |
| Physical DRAM write                   | Secure world limited to on-chip SRAM         | ×  | ×  | Undocumented memory encryption engine               | HMAC of address and data                   |
| Physical DRAM rollback write          | Secure world limited to on-chip SRAM         | ×  | ×  | Undocumented memory encryption engine               | ×  |
| Physical DRAM address reads           | Secure world in on-chip SRAM                 | ×  | ×  | ×   | ×  |
| Hardware TCB size                     | CPU chip package                             | Motherboard (CPU, TPM, DRAM, buses)        | Motherboard (CPU, TPM, DRAM, buses)        | CPU chip package                                    | CPU chip package                           |
| Software TCB size                     | Secure world OS, (firmware, application)     | All software on the computer               | SINIT ACM + VM (OS, application)           | Application module + privileged module + containers | Application module + hypervisor            |

Table 4.1 Continued: Security features overview for the trusted hardware projects related to Intel's SGX.

| Attack                                | Aegis                                      | Bastion                             | Ascend, Phantom                            | Sanctum  |
|---------------------------------------|--|-------------------------------------|--|--|
| Malicious containers (direct probing) | Security kernel separates containers       | Access checks on each memory access | OS separates containers                    | Access checks on TLB misses                          |
| Malicious OS (direct probing)         | Security kernel measured and isolated      | Memory encryption and HMAC          | ×  | Access checks on TLB misses                          |
| Malicious hypervisor (direct probing) | N/A (No hypervisor support)                | Hypervisor measured and trusted     | N/A (No hypervisor support)                | Access checks on TLB misses                          |
| Malicious firmware                    | N/A (Firmware is not active after booting) | Hypervisor measured after boot      | N/A (Firmware is not active after booting) | Firmware is measured and trusted                     |
| Malicious containers (cache timing)   | ×  | ×                                   | ×  | Each enclave gets its own cache partition            |
| Malicious OS (page fault recording)   | ×  | ×                                   | ×  | Per-enclave page tables                              |
| Malicious OS (cache timing)           | ×  | ×                                   | ×  | Non-enclave software uses a separate cache partition |
| DMA from malicious peripheral         | Equivalent to physical DRAM access         | Equivalent to physical DRAM access  | Equivalent to physical DRAM access         | MC bounces DMA outside allowed range                 |
| Physical DRAM read                    | DRAM encryption                            | DRAM encryption                     | DRAM encryption                            | ×  |
| Physical DRAM write                   | HMAC of address, data, timestamp           | Merkle tree over DRAM               | HMAC of address, data, timestamp           | ×  |
| Physical DRAM rollback write          | Merkle tree over HMAC timestamps           | Merkle tree over DRAM               | Merkle tree over HMAC timestamps           | ×  |
| Physical DRAM address reads           | ×  | ×                                   | ORAM                                       | ×  |
| Hardware TCB size                     | CPU chip package                           | CPU chip package                    | CPU chip package                           | CPU chip package                                     |
| Software TCB size                     | Application module + security kernel       | Application module + hypervisor     | Application process + trusted OS           | Application module + security monitor                |

withstand physical attacks to FIPS 140-1 Level 4 [Smith et al., 1999], and the 4765 meets the rigors of FIPS 140-2 Level 4 [nis, 2011].



**Figure 4.1:** The IBM 4765 secure coprocessor consists of an entire computer system placed inside an enclosure that can deter and detect physical attacks. The application and the system use separate processors. Sensitive memory can only be accessed by the system code, thanks to access control checks implemented in the system bus’ hardware. Dedicated hardware is used to clear the platform’s secrets and shut down the system when a physical attack is detected.

The 4765 relies heavily on physical isolation for its security properties. Its system software is protected from attacks by the application software by virtue of using a dedicated service processor that is completely separate from the application processor. Special-purpose bus logic prevents the application processor from accessing privileged resources, such as the battery-backed memory that stores the system software’s secrets.

The 4765 implements software attestation. The coprocessor’s attestation key is stored in battery-backed memory that is only accessible to the service processor. Upon reset, the service processor executes a first-stage bootloader stored in ROM, which measures and loads the system software. In turn, the system software measures the application code stored in NVRAM and loads it into the DRAM chip accessible

to the application processor. The system software provides attestation services to the application loaded inside the coprocessor.

## 4.2 ARM TrustZone

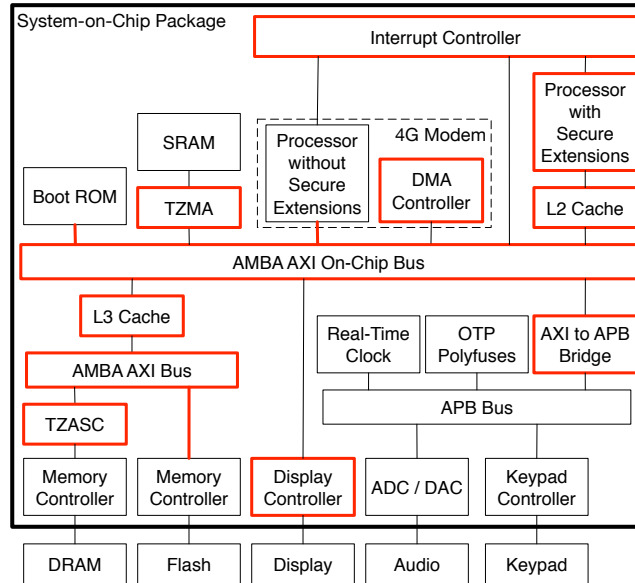
ARM's TrustZone [Alves and Felton, 2004] is a collection of hardware modules that can be used to conceptually partition a system's resources between a *secure world*, which hosts a secure container, and a *normal world*, which runs an untrusted software stack. The TrustZone documentation [ARM, 2009] describes semiconductor intellectual property cores (IP blocks) and ways in which they can be combined to achieve certain security properties, reflecting the fact that ARM is an IP core provider, not a processor manufacturer. Therefore, the mere presence of TrustZone IP blocks in a system is not sufficient to determine whether the system is secure under a specific threat model. Figure 4.2 illustrates a design for a smartphone *System-on-Chip* (SoC) design that uses TrustZone IP blocks.

TrustZone extends the address lines in the AMBA AXI system bus [ARM, 2004] with one signal that indicates whether an access belongs to the secure or normal (non-secure) world. ARM processor cores that include TrustZone's "Security Extensions" can switch between the normal world and the secure world when executing code. The address in each bus access executed by a core reflects the world in which the core is currently executing.

The reset circuitry in a TrustZone processor places it in secure mode, and points it to the first-stage bootloader stored in on-chip ROM. TrustZone's TCB includes this bootloader, which initializes the platform, sets up the TrustZone hardware to protect the secure container from untrusted software, and loads the normal world's bootloader. The secure container must also implement a monitor that performs the context switches needed to transition an execution core between the two worlds. The monitor must also handle hardware exceptions, such as interrupts, and route them to the appropriate world.

The TrustZone design gives the secure world's monitor unrestricted access to the normal world, so the monitor can implement inter-process





**Figure 4.2:** Smartphone SoC design based on TrustZone. The red IP blocks are TrustZone-aware. The red connections ignore the TrustZone secure bit in the bus address. Defining the system's security properties requires a complete understanding of all red elements in this figure.

communication (IPC) between the software in the two worlds. Specifically, the monitor can issue bus accesses using both secure and non-secure addresses. In general, the secure world's software can compromise any level in the normal world's software stack. For example, the secure container's software can jump into arbitrary locations in the normal world by flipping a bit in a register. The untrusted software in the normal world can only access the secure world via an instruction that jumps into a well-defined location inside the monitor.

Conceptually, each TrustZone CPU core provides separate address translation units for the secure and normal worlds. This is implemented by two page table base registers, and by having the page walker use the page table base corresponding to the core's current world. The physical addresses in the page table entries are extended to include the values of the secure bit to be issued on the AXI bus. The secure world is protected from untrusted software by having the CPU

core force the secure bit in the address translation result to zero for normal world address translations. As the secure container manages its own page tables, its memory accesses cannot be directly observed by the untrusted OS's page fault handler.

TrustZone-aware hardware modules, such as caches, are trusted to use the secure address bit in each bus access to enforce the isolation between worlds. For example, TrustZone's caches store the secure bit in the address tag for each cache line, which effectively provides completely different views of the memory space to the software running in different worlds. This design assumes that memory space is partitioned between the two worlds, so no aliasing can occur.

The TrustZone documentation describes two TLB configurations. If many context switches between worlds are expected, the TLB IP blocks can be configured to include the secure bit in the address tag. Alternatively, the secure bit can be omitted from the TLBs, as long as the monitor flushes the TLBs when switching contexts.

The hardware modules that do not consume TrustZone's address bit are expected to be connected to the AXI bus via IP cores that implement simple partitioning techniques. For example, the TrustZone Memory Adapter (TZMA) can be used to partition an on-chip ROM or SRAM into a secure region and a normal region, and the TrustZone Address Space Controller (TZASC) partitions the memory space provided by a DRAM controller into secure and normal regions. A TrustZone-aware DMA controller rejects DMA transfers from the normal world that reference secure world addresses.

It follows that analyzing the security properties of a TrustZone system requires a precise understanding of the behavior and configuration of all hardware modules that are attached to the AXI bus. For example, the caches described in TrustZone's documentation do not enforce a complete separation between worlds, as they allow a world's memory accesses to evict the other world's cache lines. This exposes the secure container software to cache timing attacks from the untrusted software in the normal world. Unfortunately, hardware manufacturers that license the TrustZone IP cores are reluctant to disclose all

details of their designs, making it impossible for security researchers to reason about TrustZone-based hardware.

The TrustZone components do not have any counter-measures for physical attacks. However, a system that follows the recommendations in the TrustZone documentation will not be exposed to physical attacks, under a threat model that trusts the processor package. The AXI bus is designed to connect components in an SoC design, so it cannot be tapped by an attacker. The TrustZone documentation recommends storing all secure world code and data in an on-chip SRAM, which is not assumed to be out of scope for physical attacks. However, this approach places significant limits on the secure container's functionality, because on-chip SRAM is many orders of magnitude more expensive than a DRAM module of the same capacity.

TrustZone's documentation does not describe any software attestation implementation. However, it does outline a method for implementing secure boot, which comes down to having the first-stage bootloader verify a signature in the second-stage bootloader against a public key whose cryptographic hash is burned into on-chip *One-Time Programmable* (OTP) polysilicon fuses. A hardware measurement root can be built on top of the same components, by storing a processor-specific attestation key in the polyfuses, and having the first-stage bootloader measure the second-stage bootloader and store its hash in an on-chip SRAM region allocated to the secure world. The polyfuses would be gated by a TZMA IP block that makes them accessible only to the secure world.

### 4.3 The XOM Architecture

The execute-only memory (XOM) architecture [Lie et al., 2000] introduced the approach of executing sensitive code and data in isolated containers managed by untrusted host software. XOM outlined the mechanisms needed to isolate a container's data from its untrusted software environment, such as saving the register state to a protected memory area before servicing an interrupt.

XOM supports multiple containers by tagging every cache line with the identifier of the container owning it, and ensures isolation by disallowing memory accesses to cache lines that don't match the current container's identifier. The operating system and the untrusted applications are considered to belong to a container with a null identifier.

XOM also introduced the integration of encryption and HMAC functionality in the processor's memory controller to protect container memory from physical attacks on DRAM. The encryption and HMAC functionality is used for all cache line evictions and fetches, and the ECC bits in DRAM are repurposed to store HMAC values.

XOM's design cannot guarantee DRAM freshness, so the software in its containers is vulnerable to physical replay attacks. Furthermore, XOM does not protect a container's memory access patterns, meaning that any piece of malicious software can perform cache timing attacks against the software in a container. Last, XOM containers are destroyed when they encounter hardware exceptions, such as page faults, so XOM does not support paging.

XOM predates the attestation scheme described at the beginning of the section, and relies on a modified software distribution scheme instead. Each container's contents are encrypted with a symmetric key, which also serves as the container's identity. The symmetric key, in turn, is encrypted with the public key of each CPU that is trusted to run the container. A container's author can be assured that the container is running on trusted software by embedding a secret into the encrypted container data, and using it to authenticate the container. While conceptually simpler than software attestation, this scheme does not allow the container author to vet the container's software environment.

#### **4.4 The Trusted Platform Module (TPM)**

The Trusted Platform Module (TPM) [TCG, 2003] introduced the software attestation model described at the beginning of this section. The TPM design does not require any hardware modifications to the CPU, and instead relies on an auxiliary tamper-resistant chip. The TPM module is only used to store the attestation key and to perform

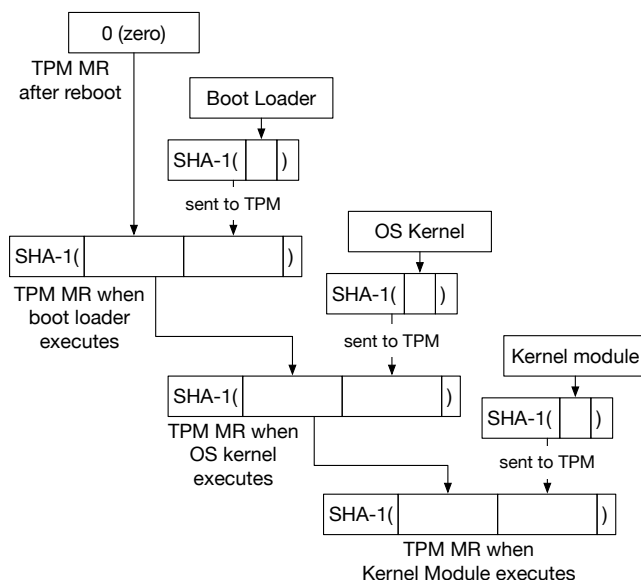
software attestation. The TPM was widely deployed on commodity computers, because it does not rely on CPU modifications. Unfortunately, the cost of this approach is that the TPM has very weak security guarantees, as explained below.

The TPM design provides one isolation container, covering all software running on the computer that has the TPM module. It follows that the measurement included in an attestation signature covers the entire OS kernel and all kernel modules, such as device drivers. However, commercial computers use a wide diversity of devices, and their system software is updated at an ever-increasing pace, so it is impossible to maintain a list of acceptable measurement hashes corresponding to a piece of trusted software. Due to this issue, the TPM's software attestation is not used in many security systems, despite its wide deployment.

The TPM design is technically not vulnerable to any software attacks, because it trusts all software on the computer. However, a TPM-based system is vulnerable to an attacker who has physical access to the machine, as the TPM module does not provide any isolation for the software on the computer. Furthermore, the TPM module receives the software measurements from the CPU, so TPM-based systems are vulnerable to attackers who can tap the communication bus between the CPU and the TPM.

Last, the TPM's design relies on the software running on the CPU to report its own cryptographic hash. The TPM module resets the measurements stored in Platform Configuration Registers (PCRs) when the computer is rebooted. Then, the TPM expects the software at each boot stage to cryptographically hash the software at the next stage, and send the hash to the TPM. The TPM updates the PCRs to incorporate the new hashes it receives, as shown in Figure 4.3. Most importantly, the PCR value at any point reflects all software hashes received by the TPM up to that point. This makes it impossible for software that has been measured to "remove" itself from the measurement.

For example, the firmware on most modern computers implements the platform initialization process in the Unified Extensible Firmware Interface (UEFI) specification [UEF, 2015]. Each platform initialization phase is responsible for verifying or measuring the firmware that im-



**Figure 4.3:** The measurement stored in a TPM platform configuration register (PCR). The PCR is reset when the system reboots. The software at every boot stage hashes the next boot stage, and sends the hash to the TPM. The PCR's new value incorporates both the old PCR value, and the new software hash.

plements the next phase. The SEC firmware initializes the TPM PCR, and then stores the PEI's measurement into a measurement register. In turn, the PEI implementation measures the DXE firmware and updates the measurement register that stores the PEI hash to account for the DXE hash. When the OS is booted, the hash in the measurement register accounts for all firmware that was used to boot the computer.

Unfortunately, the security of the whole measurement scheme hinges on the requirement that the first hash sent to the TPM must reflect the software that runs in the first boot stage. The TPM threat model explicitly acknowledges this issue, and assumes that the firmware responsible for loading the first stage bootloader is securely embedded in the motherboard. However, virtually every TPM-enabled computer stores its firmware in a flash memory module that can be reprogrammed in software (§ 2.9.1), so the TPM's measurement can be

subverted by an attacker who can re-flash the computer's firmware [Butterworth et al., 2013].

On very recent Intel processors, the attack described above can be defeated by having the initialization microcode (§ 2.14.4) hash the computer's firmware (specifically, the PEI code in UEFI [UEF, 2015] firmware) and communicate the hash to the TPM module. This is marketed as the Measured Boot feature of Intel's Boot Guard [Ruan, 2014].

Sadly, most computer manufacturers use Verified Boot (also known as "secure boot") instead of Measured Boot (also known as "trusted boot"). Verified Boot means that the processor's microcode only boots into PEI firmware that contains a signature produced by a key burned into the processor's e-fuses. Verified Boot does not impact the measurements stored on the TPM, so it does not improve the security of software attestation.

## 4.5 Intel's Trusted Execution Technology (TXT)

Intel's Trusted Execution Technology (TXT) [Grawrock, 2009] uses the TPM's software attestation model and auxiliary tamper-resistant chip, but reduces the software inside the secure container to a virtual machine (guest operating system and application) hosted by the CPU's hardware virtualization features (VMX [Uhlig et al., 2005]).

TXT isolates the software inside the container from untrusted software by ensuring that the container has exclusive control over the entire computer while it is active. This is accomplished by a secure initialization authenticated code module (SINIT ACM) that effectively performs a warm system reset before starting the container's VM.

TXT requires a TPM module with an extended register set. The registers used by the measured boot process described in § 4.4 are considered to make up the platform's Static Root of Trust Measurement (SRTM). When a TXT VM is initialized, it updates TPM registers that make up the Dynamic Root of Trust Measurement (DRTM). While the TPM's SRTM registers only reset at the start of a boot cycle, the DRTM registers are reset by the SINIT ACM, every time a TXT VM is launched.

TXT does not implement DRAM encryption or HMACs, and therefore is vulnerable to physical DRAM attacks, just like TPM-based designs. Furthermore, early TXT implementations were vulnerable to attacks where a malicious operating system would program a device, such as a network card, to perform DMA transfers to the DRAM region used by a TXT container [Wojtczuk and Rutkowska, 2009b, Wojtczuk et al., 2009]. In recent Intel CPUs, the memory controller is integrated on the CPU die, so the SINIT ACM can securely set up the memory controller to reject DMA transfers targeting TXT memory. An Intel chipset datasheet [Int, 2015c] documents an “Intel TXT DMA Protected Range” IIO configuration register.

Early TXT implementations did not measure the SINIT ACM. Instead, the microcode implementing the TXT launch instruction verified that the code module contained an RSA signature by a hard-coded Intel key. SINIT ACM signatures cannot be revoked if vulnerabilities are found, so TXT’s software attestation had to be revised when SINIT ACM exploits [Wojtczuk and Rutkowska, 2011] surfaced. Currently, the SINIT ACM’s cryptographic hash is included in the attestation measurement.

Last, the warm reset performed by the SINIT ACM does not include the software running in System Management Mode (SMM). SMM was designed solely for use by firmware, and is stored in a protected memory area (SMRAM) which should not be accessible to non-SMM software. However, the SMM handler was compromised on multiple occasions [Duflet et al., 2006, Rutkowska and Wojtczuk, 2008, Wojtczuk and Rutkowska, 2009a, Wecherowski, 2009, Embleton et al., 2010], and an attacker who obtains SMM execution can access the memory used by TXT’s container.

## **4.6 The Aegis Secure Processor**

The Aegis secure processor [Suh et al., 2003] relies on a security kernel in the operating system to isolate containers, and includes the kernel’s cryptographic hash in the measurement reported by the software attestation signature. [Suh et al., 2003] also describes a variant archi-



ture that assumes an untrusted OS. [Suh et al., 2005] argued that Physical Unclonable Functions (PUFs) [Gassend et al., 2002] can be used to endow a secure processor with a tamper-resistant private key, which is required for software attestation. PUFs do not have the fabrication process drawbacks of EEPROM, and are significantly more resilient to physical attacks than e-fuses.

Aegis relies on a trusted security kernel to isolate each container from the other software on the computer by configuring the page tables used in address translation. The security kernel is a subset of a typical OS kernel, and handles virtual memory management, processes, and hardware exceptions. As the security kernel is a part of the *trusted code base* (TCB), its cryptographic hash is included in the software attestation measurement. The security kernel uses processor features to isolate itself from the untrusted part of the operating system, such as device drivers.

The Aegis memory controller encrypts the cache lines in one memory range, and HMACs the cache lines in one other memory range. The two memory ranges can overlap, and are configurable by the security kernel. Thanks to the two ranges, the memory controller can avoid the latency overhead of cryptographic operations for the DRAM outside containers. Aegis was the first secure processor not vulnerable to physical replay attacks, as it uses a Merkle tree construction [Gassend et al., 2003] to guarantee DRAM freshness. The latency overhead of the Merkle tree is greatly reduced by augmenting the L2 cache with the tree nodes for the cache lines.

Aegis' security kernel allows the OS to page out container memory, but verifies the correctness of the paging operations. The security kernel uses the same encryption and Merkle tree algorithms as the memory controller to guarantee the confidentiality and integrity of the container pages that are swapped out from DRAM. The OS is free to page out container memory, so it can learn a container's memory access patterns, at page granularity. Aegis containers are also vulnerable to cache timing attacks.

## 4.7 The Bastion Architecture

The Bastion architecture [Champagne and Lee, 2010] introduced the use of a trusted hypervisor to provide secure containers to applications running inside unmodified, untrusted operating systems. Bastion’s hypervisor ensures that the operating system does not interfere with the secure containers. We only describe Bastion’s virtualization extensions to architectures that use nested page tables, like Intel’s VMX [Uhlig et al., 2005].

The hypervisor enforces the containers’ desired memory mappings in the OS page tables, as follows. Each Bastion container has a Security Segment that lists the virtual addresses and permissions of all pages belonging to the container, and the hypervisor maintains a Module State Table that stores an inverted page map, associating each physical memory page to its container and virtual address. The processor’s hardware page walker is modified to invoke the hypervisor on every TLB miss, before updating the TLB with the address translation result. The hypervisor checks that the virtual address used by the translation matches the expected virtual address associated with the physical address in the Module State Table.

Bastion’s cache lines are not tagged with container identifiers. Instead, only TLB entries are tagged. The hypervisor’s TLB miss handler sets the container identifier for each TLB entry as it is created. Similarly to XOM and Aegis, the secure processor checks the TLB tag against the current container’s identifier on every memory access.

Bastion offers the same protection against physical DRAM attacks as Aegis does, without the restriction that a container’s data must be stored inside a continuous DRAM range. This is accomplished by extending cache lines and TLB entries with flags that enable memory encryption and HMACing. The hypervisor’s TLB miss handler sets the flags on TLB entries, and the flags are propagated to cache lines on memory writes.

The Bastion hypervisor allows the untrusted operating system to evict secure container pages. The evicted pages are encrypted, HMACed, and covered by a Merkle tree maintained by the hypervisor. Thus, the hypervisor ensures the confidentiality, authenticity, and

freshness of the swapped pages. However, the ability to freely evict container pages allows a malicious OS to learn a container's memory accesses with page granularity. Furthermore, Bastion's threat model excludes cache timing attacks.

Bastion does not trust the platform's firmware, and computes the cryptographic hash of the hypervisor after the firmware finishes playing its part in the booting process. The hypervisor's hash is included in the measurement reported by software attestation.

## 4.8 Intel SGX

Intel's Software Guard Extensions (SGX) [McKeen et al., 2013, Anati et al., 2013, Hoekstra et al., 2013] implements secure containers for applications without making any modifications to the processor's critical execution path. SGX does not trust any layer in the computer's software stack (firmware, hypervisor, OS). Instead, SGX's TCB consists of the CPU's microcode and a few privileged containers. SGX introduces an approach to solving some of the issues raised by multi-core processors with a shared, coherent last-level cache.

SGX does not extend caches or TLBs with container identity bits, and does not require any security checks during normal memory accesses. As suggested in the TrustZone documentation, SGX always ensures that a core's TLBs only contain entries for the container that it is executing, which requires flushing the CPU core's TLBs when context-switching between containers and untrusted software.

SGX follows Bastion's approach of having the untrusted OS manage the page tables used by secure containers. The containers' security is preserved by a TLB miss handler that relies on an inverted page map (the EPCM) to reject address translations for memory that does not belong to the current container.

Like Bastion, SGX allows the untrusted operating system to evict secure container pages, in a controlled fashion. After the OS initiates a container page eviction, it must prove to the SGX implementation that it also switched the container out of all cores that were executing its code, effectively performing a very coarse-grained TLB shutdown.

SGX's microcode ensures the confidentiality, authenticity, and freshness of each container's evicted pages, like Bastion's hypervisor. However, SGX relies on a version-based Merkle tree, inspired by Aegis [Suh et al., 2003], and adds an innovative twist that allows the operating system to dynamically shape the Merkle tree. SGX also shares Bastion's and Aegis' vulnerability to memory access pattern leaks, namely a malicious OS can directly learn a container's memory accesses at page granularity, and any piece of software can perform cache timing attacks.

SGX's software attestation is implemented using Intel's Enhanced Privacy ID (EPID) group signature scheme [Brickell and Li, 2009], which is too complex for a microcode implementation. Therefore, SGX relies on an assortment of privileged containers that receive direct access to the SGX processor's hardware keys. The privileged containers are signed using an Intel private key whose corresponding public key is hard-coded into the SGX microcode, similarly to TXT's SINIT ACM.

As SGX does not protect against cache timing attacks, the privileged enclave's authors cannot use data-dependent memory accesses. For example, cache attacks on the Quoting Enclave, which computes attestation signatures, would provide an attack with a processor's EPID signing key and completely compromise SGX.

Intel's documentation states that SGX guarantees DRAM confidentiality, authentication, and freshness by virtue of a Memory Encryption Engine (MEE). The MEE is informally described in an ISCA 2015 tutorial [Int, 2015f], and in more detail in [Gueron, 2016]. It appears that SGX provides the same protection against physical DRAM attacks that Aegis and Bastion provide.

## **4.9 Sanctum**

Sanctum [Costan et al., 2015] introduced a straightforward software/hardware co-design that yields the same resilience against software attacks as SGX, and adds protection against memory access pattern leaks, such as page fault monitoring attacks and cache timing attacks.

Sanctum uses a conceptually simple cache partitioning scheme, where a computer’s DRAM is split into equally-sized continuous DRAM regions, and each DRAM region uses distinct sets in the shared last-level cache (LLC). Each DRAM region is allocated to exactly one container, so containers are isolated in both DRAM and the LLC. Containers are isolated in the other caches by flushing on context switches.

Like XOM, Aegis, and Bastion, Sanctum also considers the hypervisor, OS, and the application software to conceptually belong to a separate container. Containers are protected from the untrusted outside software by the same measures that isolate containers from each other.

Sanctum relies on a trusted security monitor, which is the first piece of firmware executed by the processor, and has the same security properties as those of Aegis’ security kernel. The monitor is measured by bootstrap code in the processor’s ROM, and its cryptographic hash is included in the software attestation measurement. The monitor verifies the operating system’s resource allocation decisions. For example, it ensures that no DRAM region is ever accessible to two different containers.

Each Sanctum container manages its own page tables mapping its DRAM regions, and handles its own page faults. It follows that a malicious OS cannot learn the virtual addresses that would cause a page fault in the container. Sanctum’s hardware modifications work in conjunction with the security monitor to make sure that a container’s page tables only reference memory inside the container’s DRAM regions.

The Sanctum design focuses completely on software attacks, and does not offer protection from any physical attack. The authors expect Sanctum’s hardware modifications to be combined with the physical attack protections in Aegis or Ascend.

## 4.10 Ascend and Phantom

The Ascend [Fletcher et al., 2012] and Phantom [Maas et al., 2013] secure processors introduced practical implementations of Oblivious RAM [Goldreich, 1987] techniques in the CPU’s memory controller. These processors are resilient to attackers who can probe the DRAM

address bus and attempt to learn a container's private information from its DRAM memory access pattern.

Implementing an ORAM scheme in a memory controller is largely orthogonal to the other secure architectures described above. It follows, for example, that Ascend's ORAM implementation can be combined with Aegis' memory encryption and authentication, and with Sanctum's hardware extensions and security monitor, yielding a secure processor that can withstand both software attacks and physical DRAM attacks.

# 5

---

## The Software Isolation Container (As Exemplified by Intel's SGX)

---

Among prior work that failed to achieve meaningful security guarantees in a realistic setting, two shortcomings are prevalent: an inability to protect against an impersonating attacker, and the inclusion of large amounts of vulnerable system software in the trusted computing base. In the context of remote computation, the system's privacy guarantees affect the system's ability to guarantee integrity: an attacker capable of learning the trusted system's secret keys can trivially defeat any protection offered by the system by emulating it (convincing the remote user that an arbitrary malicious system is the trusted system she intends to communicate with).

Another common failure is the inclusion of excessive system software in the trusted computing base. As further discussed in Section 3.5, a modern hypervisor (Xen) weighs in at 150 thousand lines of code, with the Linux kernel reaching a staggering 17 *million* lines of code. Such large code bases are (at the time of this writing) far too large for formal verification, and are dense with implementation errors. Indeed, both reveal dozens of security vulnerabilities every year, and should not be included in a trusted computing base of any security-critical application. Even if a system is able to guarantee the integrity

and privacy of a given application, the inclusion of millions of lines of buggy code in the trusted computing base makes the system's claims to security largely irrelevant.

A practically secure system trusts only the software needed to perform the security-critical task, as well as a hardware platform able to enforce its security policy and traceable to a trustworthy manufacturer. While an application cannot reasonably be secure against its own accidental or deliberate leaks of information, security-critical applications are expected to be scrutinized, and may be formally verified. A simple, easy-to-understand threat model improves the programmers' ability to write secure software, as a simple threat model simplifies the invariants the system must obey in order to be secure.

While Intel's Software Guard Extensions fall short of this ideal (as discussed in Part II of this work), the system does present a very attractive programming model: a private process with privacy and integrity guarantees assuming the software of the process itself is not vulnerable. The central concept of SGX<sup>1</sup> is the *enclave*, a protected environment that contains the code and data pertaining to a security-sensitive computation.

SGX-enabled processors provide trusted computing by isolating each enclave's environment from the untrusted software outside the enclave, and by implementing a software attestation scheme that allows a remote party to authenticate the software running inside an enclave. SGX's isolation mechanisms are intended to protect the confidentiality and integrity of the computation performed inside an enclave from attacks coming from malicious software executing on the same computer, as well as from a limited set of physical attacks.

Given that SGX is an available, documented example of an enclave-capable system, this section presents the programming model employed by the enclave primitive, as exemplified by Intel's SGX. In Part II of this work, we rely on this discussion to motivate the MIT Sanctum project, and present its hardware and software design to present a stronger security argument with an equivalent programming model.

---

<sup>1</sup> As mentioned earlier, this work discusses the original version of SGX, also referred to as SGX 1.



This section summarizes the SGX concepts that make up a mental model that is sufficient for programmers to author SGX enclaves and to add SGX support to existing system software. Unless stated otherwise, the information in this section is backed up by Intel’s Software Developer Manual (SDM). The following section builds on the concepts introduced here to fill in some of the missing pieces in the manual, and analyzes some of SGX’s security properties.

## 5.1 SGX Physical Memory Organization

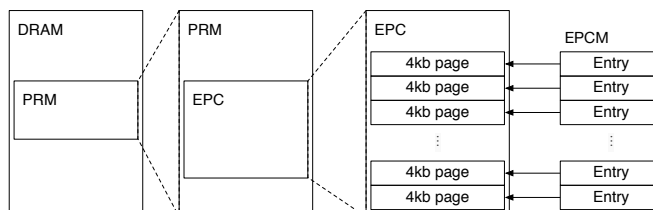
The enclaves’ code and data is stored in *Processor Reserved Memory* (PRM), which is a subset of DRAM that cannot be directly accessed by other software, including system software and SMM code. The CPU’s integrated memory controllers (§ 2.9.3) also reject DMA transfers targeting the PRM, thus protecting it from access by other peripherals.

The PRM is a continuous range of memory whose bounds are configured using a base and a mask register with the same semantics as a variable memory type range (§ 2.11.4). Therefore, the PRM’s size must be an integer power of two, and its start address must be aligned to the same power of two. Due to these restrictions, checking if an address belongs to the PRM can be done very cheaply in hardware, using the circuit outlined in § 2.11.4.

The SDM does not describe the PRM and the PRM range registers (PRMRR). These concepts are documented in the SGX manuals [Int, 2013, 2014d] and in one of the SGX papers [McKeen et al., 2013]. Therefore, the PRM is a micro-architectural detail that may change in future implementations of SGX. Our security analysis of SGX relies on implementation details surrounding the PRM, and will have to be re-evaluated for SGX future implementations.

### 5.1.1 The Enclave Page Cache (EPC)

The contents of enclaves and the associated data structures are stored in the *Enclave Page Cache* (EPC), which is a subset of the PRM, as shown in Figure 5.1.



**Figure 5.1:** Enclave data is stored into the EPC, which is a subset of the PRM. The PRM is a contiguous range of DRAM that cannot be accessed by system software or peripherals.

The SGX design supports multiple enclaves on a system concurrently, which is a necessity in multi-process environments. This is achieved by having the EPC split into 4 KB pages that can be assigned to different enclaves. The EPC uses the same page size as the architecture’s address translation feature (§ 2.5). This is not a coincidence, as future sections will reveal that the SGX implementation is tightly coupled with the address translation implementation.

The EPC is managed by the same system software that manages the rest of the computer’s physical memory. The system software, which can be a hypervisor or an OS kernel, uses SGX instructions to allocate unused pages to enclaves, and to free previously allocated EPC pages. The system software is expected to expose enclave creation and management services to application software.

Non-enclave software cannot directly access the EPC, as it is contained in the PRM. This restriction plays a key role in SGX’s enclave isolation guarantees, but creates an obstacle when the system software needs to load the initial code and data into a newly created enclave. The SGX design solves this problem by having the instructions that allocate an EPC page to an enclave also initialize the page. Most EPC pages are initialized by copying data from a non-PRM memory page.

### 5.1.2 The Enclave Page Cache Map (EPCM)

The SGX design expects the system software to allocate the EPC pages to enclaves. However, as the system software is not trusted, SGX processors check the correctness of the system software’s allocation deci-

sions, and refuse to perform any action that would compromise SGX’s security guarantees. For example, if the system software attempts to allocate the same EPC page to two enclaves, the SGX instruction used to perform the allocation will fail.

In order to perform its security checks, SGX records some information about the system software’s allocation decisions for each EPC page in the *Enclave Page Cache Map* (EPCM). The EPCM is an array with one entry per EPC page, so computing the address of a page’s EPCM entry only requires a bitwise shift operation and an addition.

The EPCM’s contents is only used by SGX’s security checks. Under normal operation, the EPCM does not generate any software-visible behavior, and enclave authors and system software developers can mostly ignore it. Therefore, the SDM only describes the EPCM at a very high level, listing the information contained within and noting that the EPCM is “trusted memory”. The SDM does not disclose the storage medium or memory layout used by the EPCM.

The EPCM uses the information in Table 5.1 to track the ownership of each EPC page. We defer a full discussion of the EPCM to a later section, because its contents is intimately coupled with all of SGX’s features, which will be described over the next few sections.

**Table 5.1:** The fields in an EPCM entry that track the ownership of pages.

| Field       | Bits | Description                            |
|-------------|------|--|
| VALID       | 1    | 0 for un-allocated EPC pages           |
| PT          | 8    | page type                              |
| ENCLAVESECS |      | identifies the enclave owning the page |

The SGX instructions that allocate an EPC page set the VALID bit of the corresponding EPCM entry to 1, and refuse to operate on EPC pages whose VALID bit is already set.

The instruction used to allocate an EPC page also determines the page’s intended usage, which is recorded in the *page type* (PT) field of the corresponding EPCM entry. The pages that store an enclave’s code and data are considered to have a *regular* type (PT\_REG in the SDM). The pages dedicated to the storage of SGX’s supporting data

structures are tagged with special types. For example, the `PT_SECS` type identifies pages that hold SGX Enclave Control Structures, which will be described in the following section. The other EPC page types will be described in future sections.

Last, a page's EPCM entry also identifies the enclave that owns the EPC page. This information is used by the mechanisms that enforce SGX's isolation guarantees to prevent an enclave from accessing another enclave's private information. As the EPCM identifies a single owning enclave for each EPC page, it is impossible for enclaves to communicate via shared memory using EPC pages. Fortunately, enclaves can share untrusted non-EPC memory, as will be discussed in § 5.2.3.

### 5.1.3 The SGX Enclave Control Structure (SECS)

SGX stores per-enclave metadata in a *SGX Enclave Control Structure* (SECS) associated with each enclave. Each SECS is stored in a dedicated EPC page with the page type `PT_SECS`. These pages are not intended to be mapped into any enclave's address space, and are exclusively used by the CPU's SGX implementation.

An enclave's identity is almost synonymous to its SECS. The first step in bringing an enclave to life allocates an EPC page to serve as the enclave's SECS, and the last step in destroying an enclave deallocates the page holding its SECS. The EPCM entry field identifying the enclave that owns an EPC page points to the enclave's SECS. The system software uses the virtual address of an enclave's SECS to identify the enclave when invoking SGX instructions.

All SGX instructions take virtual addresses as their inputs. Given that SGX instructions use SECS addresses to identify enclaves, the system software must create entries in its page tables pointing to the SECS of the enclaves it manages. However, the system software cannot access any SECS page, as these pages are stored in the PRM. SECS pages are not intended to be mapped inside their enclaves' virtual address spaces, and SGX-enabled processors explicitly prevent enclave code from accessing SECS pages.

This seemingly arbitrary limitation is in place so that the SGX implementation can store sensitive information in the SECS, and be

able to assume that no potentially malicious software will access that information. For example, the SDM states that each enclave's measurement is stored in its SECS. If software would be able to modify an enclave's measurement, SGX's software attestation scheme would provide no security assurances.

The SECS is strongly coupled with many of SGX's features. Therefore, the pieces of information that make up the SECS will be gradually introduced as the different aspects of SGX are described.

## 5.2 The Memory Layout of an SGX Enclave

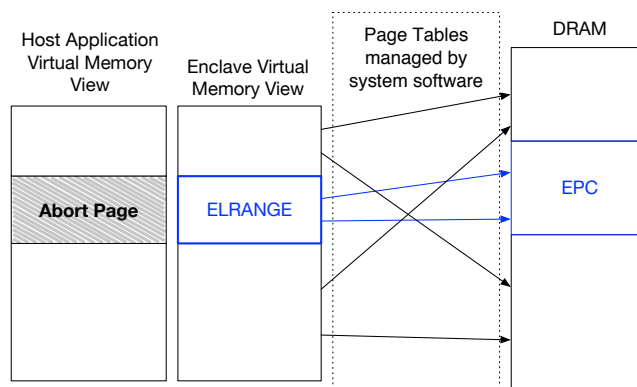
SGX was designed to minimize the effort required to convert application code to take advantage of enclaves. History suggests this is a wise decision, as a large factor in the continued dominance of the Intel architecture is its ability to maintain backward compatibility. To this end, SGX enclaves were designed to be conceptually similar to the leading software modularization construct, dynamically loaded libraries, which are packaged as `.so` files on Unix, and `.dll` files on Windows.

For simplicity, we describe the interaction between enclaves and non-enclave software assuming that each enclave is used by exactly one application process, which we shall refer to as the enclave's *host process*. We do note, however, that the SGX design does not explicitly prohibit multiple application processes from sharing an enclave.

### 5.2.1 The Enclave Linear Address Range (ELRANGE)

Each enclave designates an area in its virtual address space, called the *enclave linear address range* (ELRANGE), which is used to map the code and the sensitive data stored in the enclave's EPC pages. The virtual address space outside ELRANGE is mapped to access non-EPC memory via the same virtual addresses as the enclave's host process, as shown in Figure 5.2.

The SGX design guarantees that the enclave's memory accesses inside ELRANGE obey the virtual memory abstraction (§ 2.5.1), while memory accesses outside ELRANGE receive no guarantees. Therefore, enclaves must store all of their code and private data inside EL-



**Figure 5.2:** An enclave’s EPC pages are accessed using a dedicated region in the enclave’s virtual address space, called ELRANGE. The rest of the virtual address space is used to access the memory of the host process. The memory mappings are established using the page tables managed by system software.

RANGE, and must consider the memory outside ELRANGE to be an untrusted interface to the outside world.

The word “linear” in ELRANGE references the linear addresses produced by the vestigial segmentation feature (§ 2.7) in the 64-bit Intel architecture. For most purposes, “linear” can be treated as a synonym for “virtual”.

ELRANGE is specified using a base (the BASEADDR field) and a size (the SIZE) in the enclave’s SECS (§ 5.1.3). ELRANGE must meet the same constraints as a variable memory type range (§ 2.11.4) and as the PRM range (§ 5.1), namely the size must be a power of 2, and the base must be aligned to the size. These restrictions are in place so that the SGX implementation can inexpensively check whether an address belongs to an enclave’s ELRANGE, in either hardware (§ 2.11.4) or software.

When an enclave represents a dynamic library, it is natural to set ELRANGE to the memory range reserved for the library by the loader. The ability to access non-enclave memory from enclave code makes it easy to reuse existing library code that expects to work with pointers to memory buffers managed by code in the host process.

Non-enclave software cannot access PRM memory. A memory access that resolves inside the PRM results in an aborted transaction, which is undefined at an architectural level. On current processors, aborted writes are ignored, and aborted reads return a value whose bits are all set to 1. This comes into play in the scenario described above, where an enclave is loaded into a host application process as a dynamically loaded library. The system software maps the enclave's code and data in ELRANGE into EPC pages. If application software attempts to access memory inside ELRANGE, it will experience the abort transaction semantics. The current semantics do not cause the application to crash (e.g., due to a Page Fault), but also guarantee that the host application will not be able to tamper with the enclave or read its private information.

### 5.2.2 SGX Enclave Attributes

The execution environment of an enclave is heavily influenced by the value of the ATTRIBUTES field in the enclave's SECS (§ 5.1.3). The rest of this work will refer to the field's sub-fields, shown in Table 5.2, as *enclave attributes*.

**Table 5.2:** An enclave's attributes are the sub-fields in the ATTRIBUTES field of the enclave's SECS. This table shows a subset of the attributes defined in the SGX documentation.

| Field     | Bits | Description  |
|-----------|------|--|
| DEBUG     | 1    | Opts into enclave debugging features.                            |
| XFRM      | 64   | The value of XCR0 (§ 2.6) while this enclave's code is executed. |
| MODE64BIT | 1    | Set for 64-bit enclaves.   |

The most important attribute, from a security perspective, is the DEBUG flag. When this flag is set, it enables the use of SGX's debugging features for this enclave. These debugging features include the ability to read and modify most of the enclave's memory. Therefore, DEBUG should only be set in a development environment, as it causes the enclave to lose all SGX security guarantees.

SGX guarantees that enclave code will always run with the XCR0 register (§ 2.6) set to the value indicated by *extended features request mask* (XFRM). Enclave authors are expected to use XFRM to specify the set of architectural extensions enabled by the compiler used to produce the enclave's code. Having XFRM be explicitly specified allows Intel to design new architectural extensions that change the semantics of existing instructions, such as Memory Protection Extensions (MPX), without having to worry about the security implications on enclave code that was developed without an awareness of the new features.

The MODE64BIT flag is set to true for enclaves that use the 64-bit Intel architecture. From a security standpoint, this flag should not even exist, as supporting a secondary architecture adds unnecessary complexity to the SGX implementation, and increases the probability that security vulnerabilities will creep in. It is very likely that the 32-bit architecture support was included due to Intel's strategy of offering extensive backwards compatibility, which has paid off quite well so far.

In the interest of mental sanity, this work does not analyze the behavior of SGX for enclaves whose MODE64BIT flag is cleared. However, a security researcher who wishes to find vulnerabilities in SGX may study this area.

Last, the INIT flag is always false when the enclave's SECS is created. The flag is set to true at a certain point in the enclave lifecycle, which will be summarized in § 5.3.

### 5.2.3 Address Translation for SGX Enclaves

Under SGX, the operating system and hypervisor are still in full control of the page tables and EPTs, and each enclave's code uses the same address translation process and page tables (§ 2.5) as its host application. This minimizes the amount of changes required to add SGX support to existing system software. At the same time, having the page tables managed by untrusted system software opens SGX up to the address translation attacks described in § 3.7. As future sections will reveal, a good amount of the complexity in SGX's design can be attributed to the need to prevent these attacks.



SGX's active memory mapping attacks defense mechanisms revolve around ensuring that each EPC page can only be mapped at a specific virtual address (§ 2.7). When an EPC page is allocated, its intended virtual address is recorded in the EPCM entry for the page, in the ADDRESS field.

When an address translation (§ 2.5) result is the physical address of an EPC page, the CPU ensures<sup>2</sup> that the virtual address given to the address translation process matches the expected virtual address recorded in the page's EPCM entry.

SGX also protects against some passive memory mapping attacks and fault injection attacks by ensuring that the access permissions of each EPC page always match the enclave author's intentions. The access permissions for each EPC page are specified when the page is allocated, and recorded in the *readable* (R), *writable* (W), and *executable* (X) fields in the page's EPCM entry, shown in Table 5.3.

**Table 5.3:** The fields in an EPCM entry that indicate the enclave's intended virtual memory layout.

| Field   | Bits | Description   |
|---------|------|---|
| ADDRESS | 48   | the virtual address used to access this page            |
| R       | 1    | allow reads by enclave code                             |
| W       | 1    | allow writes by enclave code                            |
| X       | 1    | allow execution of code inside the page, inside enclave |

When an address translation (§ 2.5) resolves into an EPC page, the corresponding EPCM entry's fields override the access permission attributes (§ 2.5.3) specified in the page tables. For example, the W field in the EPCM entry overrides the writable (W) attribute, and the X field overrides the disable execution (XD) attribute.

It follows that an enclave author must include memory layout information along with the enclave, in such a way that the system software loading the enclave will know the expected virtual memory address and access permissions for each enclave page. In return, the

<sup>2</sup>A mismatch triggers a general protection fault (#GP, § 2.8.2).

SGX design guarantees to the enclave authors that the system software, which manages the page tables and EPT, will not be able to set up an enclave's virtual address space in a manner that is inconsistent with the author's expectations.

The `.so` and `.dll` file formats, which are SGX's intended enclave delivery vehicles, already have provisions for specifying the virtual addresses that a software module was designed to use, as well as the desired access permissions for each of the module's memory areas.

Last, a SGX-enabled CPU will ensure that the virtual memory inside ELRANGE (§ 5.2.1) is mapped to EPC pages. This prevents the system software from carrying out an address translation attack where it maps the enclave's entire virtual address space to DRAM pages outside the PRM, which do not trigger any of the checks above, and can be directly accessed by the system software.

#### **5.2.4 The Thread Control Structure (TCS)**

The SGX design fully embraces multi-core processors. It is possible for multiple logical processors (§ 2.9.3) to concurrently execute the same enclave's code concurrently, via different threads.

The SGX implementation uses a *Thread Control Structure* (TCS) for each logical processor that executes an enclave's code. It follows that an enclave's author must provision at least as many TCS instances as the maximum number of concurrent threads that the enclave is intended to support.

Each TCS is stored in a dedicated EPC page whose EPCM entry type is `PT_TCS`. The SDM describes the first few fields in the TCS. These fields are considered to belong to the architectural part of the structure, and therefore are guaranteed to have the same semantics on all processors that support SGX. The rest of the TCS is not documented.

The contents of an EPC page that holds a TCS cannot be directly accessed, even by the code of the enclave that owns the TCS. This restriction is similar to the restriction on accessing EPC pages holding SECS instances. However, the architectural fields in a TCS can be read by enclave debugging instructions.

The architectural fields in the TCS lay out the context switches (§ 2.6) performed by a logical processor when it transitions between executing non-enclave and enclave code.

For example, the OENTRY field specifies the value loaded in the instruction pointer (RIP) when the TCS is used to start executing enclave code, so the enclave author has strict control over the entry points available to enclave's host application. Furthermore, the OFSBASGX and OFSBASGX fields specify the base addresses loaded in the FS and GS segment registers (§ 2.7), which typically point to Thread Local Storage (TLS).

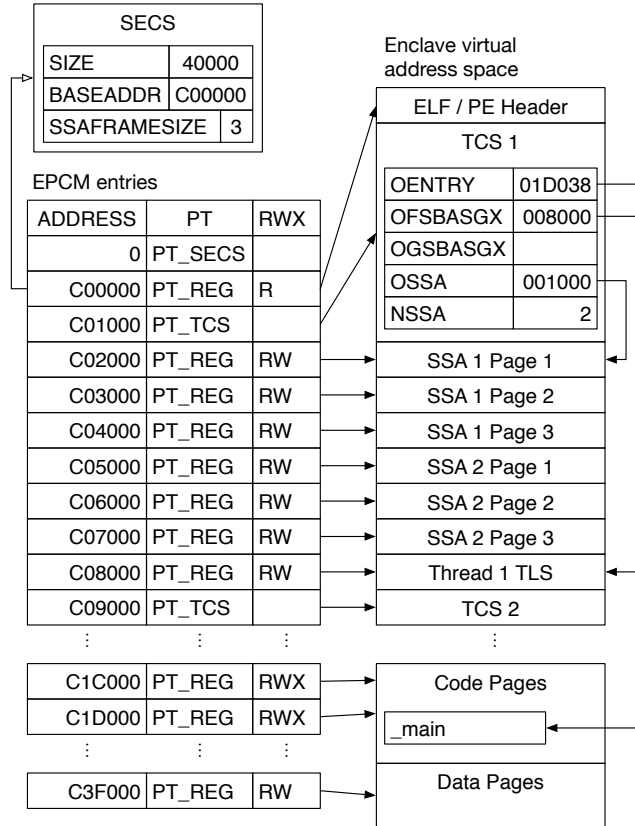
### 5.2.5 The State Save Area (SSA)

When the processor encounters a hardware exception (§ 2.8.2), such as an interrupt (§ 2.12), while executing the code inside an enclave, it performs a privilege level switch (§ 2.8.2) and invokes a hardware exception handler provided by the system software. Before executing the exception handler, however, the processor needs a secure area to store the enclave code's execution context (§ 2.6), so that the information in the execution context is not revealed to the untrusted system software.

In the SGX design, the area used to store an enclave thread's execution context while a hardware exception is handled is called a **State Save Area (SSA)**, illustrated in Figure 5.3. Each TCS references a contiguous sequence of SSAs. The *offset of the SSA array* (OSSA) field specifies the location of the first SSA in the enclave's virtual address space. The *number of SSAs* (NSSA) field indicates the number of available SSAs.

Each SSA starts at the beginning of an EPC page, and uses up the number of EPC pages that is specified in the SSAFRAMESIZE field of the enclave's SECS. These alignment and size restrictions most likely simplify the SGX implementation by reducing the number of special cases that it needs to handle.

An enclave thread's execution context consists of the general-purpose registers (GPRs) and the result of the XSAVE instruction (§ 2.6). Therefore, the size of the execution context depends on the requested-feature bitmap (RFBM) used by XSAVE. All code



**Figure 5.3:** A possible layout of an enclave's virtual address space. Each enclave has a SECS, and one TCS per supported concurrent thread. Each TCS points to a sequence of SSAs, and specifies initial values for RIP and for the base addresses of FS and GS.

in an enclave uses the same RFBM, which is declared in the XFRM enclave attribute (§ 5.2.2). The number of EPC pages reserved for each SSA, specified in SSAFRAME SIZE, must<sup>3</sup> be large enough to fit the XSAVE output for the feature bitmap specified by XFRM.

SSAs are stored in regular EPC pages, whose EPCM page type is PT\_REG. Therefore, the SSA contents is accessible to enclave software. The SSA layout is architectural, and is completely documented

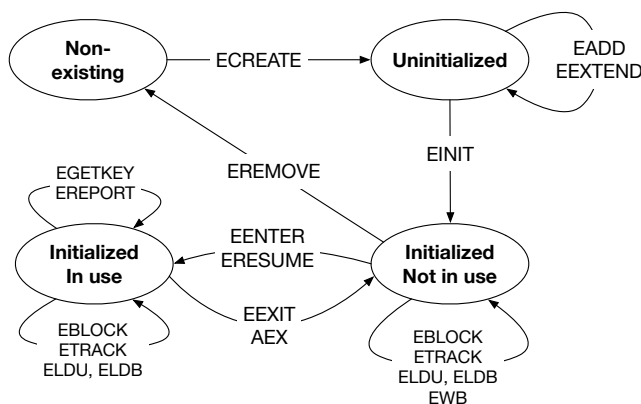
<sup>3</sup>ECREATE (§ 5.3.1) fails if SSAFRAME SIZE is too small.

in the SDM. This opens up possibilities for an enclave exception handler that is invoked by the host application after a hardware exception occurs, and acts upon the information in a SSA.

### 5.3 The Life Cycle of an SGX Enclave

An enclave's life cycle is deeply intertwined with resource management, specifically the allocation of EPC pages. Therefore, the instructions that transition between different life cycle states can only be executed by the system software. The system software is expected to expose the SGX instructions described below as enclave loading and teardown services.

The following subsections describe the major steps in an enclave's lifecycle, which is illustrated by Figure 5.4.



**Figure 5.4:** The SGX enclave life cycle management instructions and state transition diagram.

#### 5.3.1 Creation

An enclave is born when the system software issues the **ECREATE** instruction, which turns a free EPC page into the SECS (§ 5.1.3) for the new enclave.

**ECREATE** initializes the newly created SECS using the information in a non-EPC page owned by the system software. This page specifies

the values for all SECS fields defined in the SDM, such as `BASEADDR` and `SIZE`, using an architectural layout that is guaranteed to be preserved by future implementations.

While it is very likely that the actual SECS layout used by initial SGX implementations matches the architectural layout quite closely, future implementations are free to deviate from this layout, as long as they maintain the ability to initialize the SECS using the architectural layout. Software cannot access an EPC page that holds a SECS, so it cannot become dependent on an internal SECS layout. This is a stronger version of the encapsulation used in the Virtual Machine Control Structure (VMCS, § 2.8.3).

`ECREATE` validates the information used to initialize the SECS, and results in a page fault (`#PF`, § 2.8.2) or general protection fault (`#GP`, § 2.8.2) if the information is not valid. For example, if the `SIZE` field is not a power of two, `ECREATE` results in `#GP`. This validation, combined with the fact that the SECS is not accessible by software, simplifies the implementation of the other SGX instructions, which can assume that the information inside the SECS is valid.

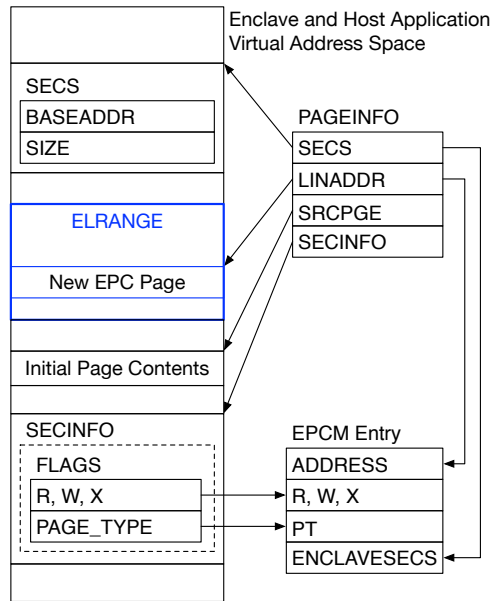
Last, `ECREATE` initializes the enclave's `INIT` attribute (sub-field of the `ATTRIBUTES` field in the enclave's SECS, § 5.2.2) to the false value. The enclave's code cannot be executed until the `INIT` attribute is set to true, which happens in the initialization stage that will be described in § 5.3.3.

### 5.3.2 Loading

`ECREATE` marks the newly created SECS as *uninitialized*. While an enclave's SECS is in this state, the system software can use `EADD` instructions to load the initial code and data into the enclave. `EADD` is used to create both TCS pages (§ 5.2.4) and regular pages.

`EADD` reads its input data from a *Page Information* (`PAGEINFO`) structure, illustrated in Figure 5.5. The structure's contents are only used to communicate information to the SGX implementation, so it is entirely architectural and documented in the SDM.

Currently, the `PAGEINFO` structure contains the virtual address of the EPC page that will be allocated (`LINADDR`), the virtual address



**Figure 5.5:** The PAGEINFO structure supplies input data to SGX instructions such as EADD.

of the non-EPC page whose contents will be copied into the newly allocated EPC page (SRCPGE), a virtual address that resolves to the SECS of the enclave that will own the page (SECS), and values for some of the fields of the EPCM entry associated with the newly allocated EPC page (SECINFO).

The SECINFO field in the PAGEINFO structure is actually a virtual memory address, and points to a *Security Information* (SECINFO) structure, some of which is also illustrated in Figure 5.5. The SECINFO structure contains the newly allocated EPC page's access permissions (R, W, X) and its EPCM page type (PT\_REG or PT\_TCS). Like PAGEINFO, the SECINFO structure is solely used to communicate data to the SGX implementation, so its contents are also entirely architectural. However, most of the structure's 64 bytes are reserved for future use.

Both the PAGEINFO and the SECINFO structures are prepared by the system software that invokes the EADD instruction, and there-

fore must be contained in non-EPC pages. Both structures must be aligned to their sizes – PAGEINFO is 32 bytes long, so each PAGEINFO instance must be 32-byte aligned, while SECINFO has 64 bytes, and therefore each SECINFO instance must be 64-byte aligned. The alignment requirements likely simplify the SGX implementation by reducing the number of special cases that must be handled.

EADD validates its inputs before modifying the newly allocated EPC page or its EPCM entry. Most importantly, attempting to EADD a page to an enclave whose SECS is in the initialized state will result in a #GP. Furthermore, attempting to EADD an EPC page that is already allocated (the VALID field in its EPCM entry is 1) results in a #PF. EADD also ensures that the page's virtual address falls within the enclave's ELRANGE, and that all reserved fields in SECINFO are set to zero.

While loading an enclave, the system software will also use the EEXTEND instruction, which updates the enclave's measurement used in the software attestation process. Software attestation is discussed in § 5.8.

### 5.3.3 Initialization

After loading the initial code and data pages into the enclave, the system software must use a *Launch Enclave* (LE) to obtain an EINIT Token Structure, via an under-documented process that will be described in more detail in § 5.9.1. The token is then provided to the EINIT instruction, which marks the enclave's SECS as *initialized*.

The LE is a privileged enclave provided by Intel, and **is a prerequisite for the use of enclaves authored by parties other than Intel**. The LE is an SGX enclave, so it must be created, loaded and initialized using the processes described in this section. However, the LE is cryptographically signed (§ 3.1.3) with a special Intel key that is hard-coded into the SGX implementation, and that causes EINIT to initialize the LE without checking for a valid EINIT Token Structure.

When EINIT completes successfully, it sets the enclave's INIT attribute to true. This opens the way for ring 3 (§ 2.3) application software to execute the enclave's code, using the SGX instructions described in § 5.4. On the other hand, once INIT is set to true, EADD



cannot be invoked on that enclave anymore, so the system software must load all pages that make up the enclave's initial state before executing the `EINIT` instruction.

#### 5.3.4 Teardown

After the enclave has done the computation it was designed to perform, the system software executes the `EREMOVE` instruction to deallocate the EPC pages used by the enclave.

`EREMOVE` marks an EPC page as available by setting the `VALID` field of the page's EPCM entry to 0 (zero). Before freeing up the page, `EREMOVE` makes sure that there is no logical processor executing code inside the enclave that owns the page to be removed.

An enclave is completely destroyed when the EPC page holding its SECS is freed. `EREMOVE` refuses to deallocate a SECS page if it is referenced by any other EPCM entry's `ENCLAVESECS` field, so an enclave's SECS page can only be deallocated after all pages belonging to the enclave have been deallocated.

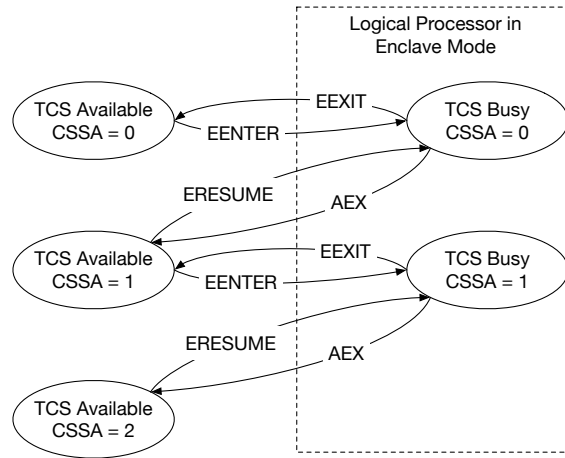
### 5.4 The Life Cycle of an SGX Thread

Between the time when an enclave is initialized (§ 5.3.3) and the time when it is torn down (§ 5.3.4), the enclave's code can be executed by any application process that has the enclave's EPC pages mapped into its virtual address space.

When executing the code inside an enclave, a logical processor is said to be *in enclave mode*, and the code that it executes can access the regular (`PT_REG`, § 5.1.2) EPC pages that belong to the currently executing enclave. When a logical process is outside enclave mode, it bounces any memory accesses inside the Processor Reserved Memory range (`PRM`, § 5.1), which includes the EPC.

Each logical processor that executes enclave code uses a Thread Control Structure (`TCS`, § 5.2.4). When a `TCS` is used by a logical processor, it is said to be *busy*, and it cannot be used by any other logical processor. Figure 5.6 illustrates the instructions used by

a host process to execute enclave code and their interactions with the TCS that they target.



**Figure 5.6:** The stages of the life cycle of an SGX Thread Control Structure (TCS) that has two State Save Areas (SSAs).

Assuming that no hardware exception occurs, an enclave's host process uses the **EENTER** instruction, described in § 5.4.1, to execute enclave code. When the enclave code finishes performing its task, it uses the **EEXIT** instruction, covered in § 5.4.2, to return the execution control to the host process that invoked the enclave.

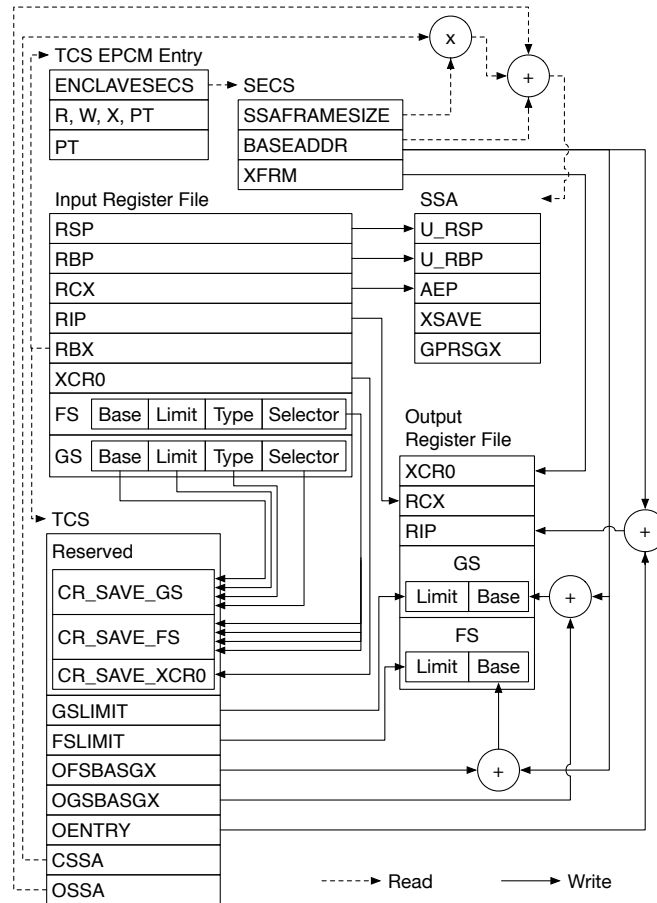
If a hardware exception occurs while a logical processor is in enclave mode, the processor is taken out of enclave mode using an *Asynchronous Enclave Exit* (AEX), summarized in § 5.4.3, before the system software's exception handler is invoked. After the system software's handler is invoked, the enclave's host process can use the **ERESUME** instruction, described in § 5.4.4, to re-enter the enclave and resume the computation that it was performing.

#### 5.4.1 Synchronous Enclave Entry

At a high level, **EENTER** performs a controlled jump into enclave code, while performing the processor configuration that is needed by SGX's security guarantees. Going through all configuration steps is

a tedious exercise, but is a necessary prerequisite to understanding how all data structures used by SGX work together. For this reason, **EENTER** and its siblings are described in much more detail than the other SGX instructions.

**EENTER**, illustrated in Figure 5.7 can only be executed by unprivileged application software running at ring 3 (§ 2.3), and results in an undefined instruction (**#UD**) fault if it is executed by system software.



**Figure 5.7:** Data flow diagram for a subset of the logic in **EENTER**. The figure omits the logic for disabling debugging features, such as hardware breakpoints and performance monitoring events.

**EENTER** switches the logical processor to enclave mode, but does not perform a privilege level switch (§ 2.8.2). Therefore, enclave code always executes at ring 3, with the same privileges as the application code that calls it. This makes it possible for an infrastructure owner to allow user-supplied software to create and use enclaves, while having the assurance that the OS kernel and hypervisor can still protect the infrastructure from buggy or malicious software.

**EENTER** takes the virtual address of a TCS as its input, and requires that the TCS is *available* (not busy), and that at least one State Save Area (SSA, § 5.2.5) is available in the TCS. The latter check is implemented by making sure that the *current SSA index* (CSSA) field in the TCS is less than the number of SSAs (NSSA) field. The SSA indicated by the CSSA, which shall be called the *current SSA*, is used in the event that a hardware exception occurs while enclave code is executed.

**EENTER** transitions the logical processor into enclave mode, and sets the instruction pointer (RIP) to the value indicated by the *entry point offset* (OENTRY) field in the TCS that it receives. **EENTER** is used by an untrusted caller to execute code in a protected environment, and therefore has the same security considerations as **SYSCALL** (§ 2.8), which is used to call into system software. Setting RIP to the value indicated by OENTRY guarantees to the enclave author that the enclave code will only be invoked at well defined points, and prevents a malicious host application from bypassing any security checks that the enclave author may perform.

**EENTER** also sets XCR0 (§ 2.6), the register that controls which extended architectural features are in use, to the value of the XFRM enclave attribute (§ 5.2.2). Ensuring that XCR0 is set according to the enclave author's intentions prevents a malicious operating system from bypassing an enclave's security by enabling architectural features that the enclave is not prepared to handle.

Furthermore, **EENTER** loads the bases of the segment registers (§ 2.7) FS and GS using values specified in the TCS. The segments' selectors and types are hard-coded to safe values for ring 3 data segments. This aspect of the SGX design makes it easy to implement per-thread Thread Local Storage (TLS). For 64-bit enclaves, this is a convenience feature

rather than a security measure, as enclave code can securely load new bases into FS and GS using the `WRFSBASE` and `WRGSBASE` instructions.

The `EENTER` implementation backs up the old values of the registers that it modifies, so they can be restored when the enclave finishes its computation. Just like `SYSCALL`, `EENTER` saves the address of the following instruction in the RCX register.

Interestingly, the SDM states that the old values of the XCR0, FS, and GS registers are saved in new registers dedicated to the SGX implementation. However, given that they will only be used on an enclave exit, we expect that the registers are saved in DRAM, in the reserved area in the TCS.

Like `SYSCALL`, `EENTER` does not modify the stack pointer register (RSP). To avoid any security exploits, enclave code should set RSP to point to a stack area that is entirely contained in EPC pages. Multi-threaded enclaves can easily implement per-thread stack areas by setting up each thread's TLS area to include a pointer to the thread's stack, and by setting RSP to the value obtained by reading the TLS area at which the FS or GS segment points.

Last, when `EENTER` enters enclave mode, it suspends some of the processor's debugging features, such as hardware breakpoints and Precise Event Based Sampling (PEBS). Conceptually, a debugger attached to the host process sees the enclave's execution as one single processor instruction.

#### 5.4.2 Synchronous Enclave Exit

`EEXIT` can only be executed while the logical processor is in enclave mode, and results in a (`#UD`) if executed in any other circumstances. In a nutshell, the instruction returns the processor to ring 3 outside enclave mode and restores the registers saved by `EENTER`, which were described above.

Unlike `SYSRET`, `EEXIT` sets RIP to the value read from RBX, after exiting enclave mode. This is inconsistent with `EENTER`, which saves the RIP value to RCX. Unless this inconsistency stems from an error in the SDM, enclave code must be sure to note the difference.

The SDM explicitly states that **EEXIT** does not modify most registers, so enclave authors must make sure to clear any secrets stored in the processor's registers before returning control to the host process. Furthermore, enclave software will most likely cause a fault in its caller if it doesn't restore the stack pointer **RSP** and the stack frame base pointer **RBP** to the values that they had when **EENTER** was called.

It may seem unfortunate that enclave code can induce faults in its caller. For better or for worse, this perfectly matches the case where an application calls into a dynamically loaded module. More specifically, the module's code is also responsible for preserving stack-related registers, and a buggy module may jump into any address in the application code of the host process.

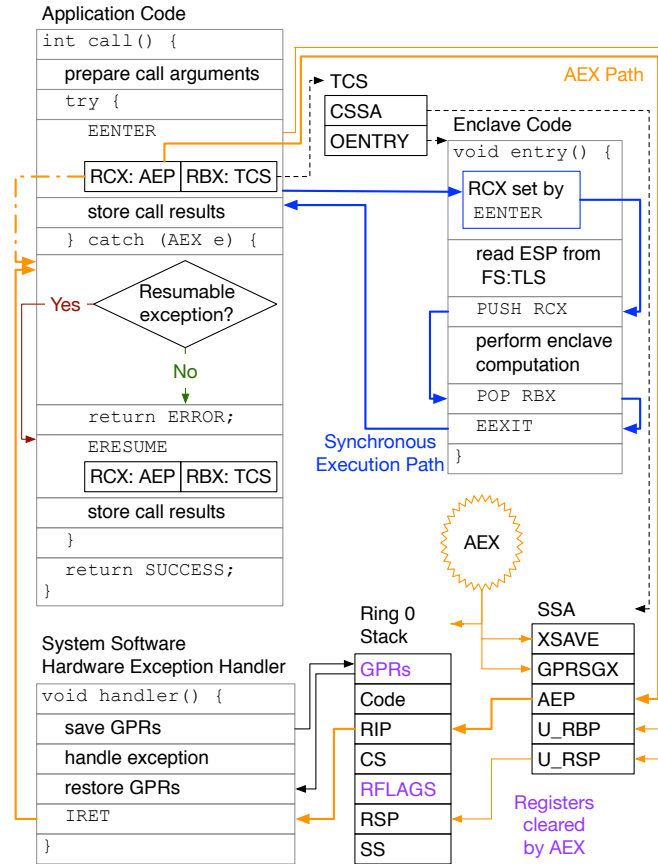
This section describes the **EENTER** behavior for 64-bit enclaves. The **EENTER** implementation for 32-bit enclaves is significantly more complex, due to the extra special cases introduced by the full-fledged segmentation model that is still present in the 32-bit Intel architecture. As stated in the introduction, we are not interested in such legacy aspects.

### 5.4.3 Asynchronous Enclave Exit (AEX)

If a hardware exception, like a fault (§ 2.8.2) or an interrupt (§ 2.12), occurs while a logical processor is executing an enclave's code, the processor performs an *Asynchronous Enclave Exit* (AEX) before invoking the system software's exception handler, as shown in Figure 5.8.

The AEX saves the enclave code's execution context (§ 2.6), restores the state saved by **EENTER**, and sets up the processor registers so that the system software's hardware exception handler will return to an *asynchronous exit handler* in the enclave's host process. The exit handler is expected to use the **ERESUME** instruction to resume the enclave computation that was interrupted by the hardware exception.

Asides from the behavior described in § 5.4.1, **EENTER** also writes some information to the current SSA, which is only used if an AEX occurs. As shown in Figure 5.7, **EENTER** stores the stack pointer register **RSP** and the stack frame base pointer register **RBP** into the **U\_RSP** and **U\_RBP** fields in the current SSA. Last, **EENTER** stores



**Figure 5.8:** If a hardware exception occurs during enclave execution, the synchronous execution path is aborted, and an Asynchronous Enclave Exit (AEX) occurs instead.

the value in RCX in the *Asynchronous Exit handler Pointer* (AEP) field in the current SSA.

When a hardware exception occurs in enclave mode, the SGX implementation performs a sequence of steps that takes the logical processor out of enclave mode and invokes the hardware exception handler in the system software. Conceptually, the SGX implementation first performs an AEX to take the logical processor out of enclave mode, and then the hardware exception is handled using the standard Intel architecture's

behavior described in § 2.8.2. Actual Intel processors may interleave the AEX implementation with the exception handling implementation. However, for simplicity, this work describes AEX as a separate process that is performed before any exception handling steps are taken.

In the Intel architecture, if a hardware exception occurs, the application code's execution context can be read and modified by the system software's exception handler (§ 2.8.2). This is acceptable when the system software is trusted by the application software. However, under SGX's threat model, the system software is not trusted by enclaves. Therefore, the AEX step erases any secrets that may exist in the execution state by resetting all its registers to predefined values.

Before the enclave's execution state is reset, it is backed up inside the current SSA. Specifically, an AEX backs up the general purpose registers (GPRs, § 2.6) in the GPRSGX area in the SSA, and then performs an **XSAVE** (§ 2.6) using the requested-feature bitmap (RFBM) specified in the **XFRM** field in the enclave's SECS. As each SSA is entirely stored in EPC pages allocated to the enclave, the system software cannot read or tamper with the backed up execution state. When an SSA receives the enclave's execution state, it is marked as used by incrementing the **CSSA** field in the current TCS.

After clearing the execution context, the AEX process sets **RSP** and **RBP** to the values saved by **EENTER** in the current SSA, and sets **RIP** to the value in the current SSA's **AEP** field. This way, when the system software's hardware exception handler completes, the processor will execute the asynchronous exit handler code in the enclave's host process. The SGX design makes it easy to set up the asynchronous handler code as an exception handler in the routine that contains the **EENTER** instruction, because the **RSP** and **RBP** registers will have the same values as they had when **EENTER** was executed.

Many of the actions taken by AEX to get the logical processor outside of enclave mode match **EEXIT**. The segment registers **FS** and **GS** are restored to the values saved by **EENTER**, and all debugging facilities that were suppressed by **EENTER** are restored to their previous states.



#### 5.4.4 Recovering from an Asynchronous Exit

When a hardware exception occurs inside enclave mode, the processor performs an AEX before invoking the exception's handler set up by the system software. The AEX sets up the execution context in such a way that when the system software finishes processing the exception, it returns into an asynchronous exit handler in the enclave's host process. The asynchronous exception handler usually executes the `ERESUME` instruction, which causes the logical processor to go back into enclave mode and continue the computation that was interrupted by the hardware exception.

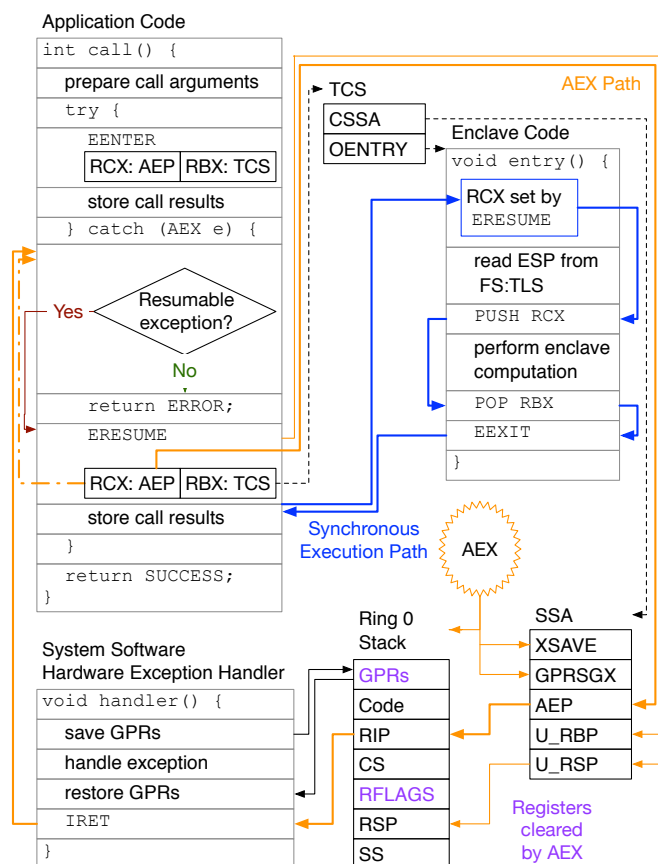
`ERESUME` shares much of its functionality with `EENTER`. This is best illustrated by the similarity between Figures 5.9 and 5.8.

`EENTER` and `ERESUME` receive the same inputs, namely a pointer to a TCS, described in § 5.4.1, and an AEP, described in § 5.4.3. The most common application design will pair each `EENTER` instance with an asynchronous exit handler that invokes `ERESUME` with exactly the same arguments.

The main difference between `ERESUME` and `EENTER` is that the former uses an SSA that was “filled out” by an AEX (§ 5.4.3), whereas the latter uses an empty SSA. Therefore, `ERESUME` results in a `#GP` fault if the CSSA field in the provided TCS is 0 (zero), whereas `EENTER` fails if CSSA is greater than or equal to NSSA.

When successful, `ERESUME` decrements the CSSA field of the TCS, and restores the execution context backed up in the SSA pointed to by the CSSA field in the TCS. Specifically, the `ERESUME` implementation restores the GPRs (§ 2.6) from the GPRSGX field in the SSA, and performs an `XRSTOR` (§ 2.6) to load the execution state associated with the extended architectural features used by the enclave.

`ERESUME` shares the following behavior with `EENTER` (§ 5.4.1). Both instructions write the `U_RSP`, `U_RBP`, and `AEP` fields in the current SSA. Both instructions follow the same process for backing up `XCR0` and the `FS` and `GS` segment registers, and set them to the same values, based on the current TCS and its enclave's `SECS`. Last, both instructions disable the same subset of the logical processor's debugging features.



**Figure 5.9:** If a hardware exception occurs during enclave execution following an `ERESUME`, the synchronous execution path is aborted, and an Asynchronous Enclave Exit (AEX) occurs instead.

An interesting edge case that `ERESUME` handles correctly is that it sets `XCR0` to the XFRM enclave attribute **before** performing an `XRSTOR`. It follows that `ERESUME` fails if the requested feature bitmap (RFBM) in the SSA is not a subset of XFRM. This matters because, while an AEX will always use the XFRM value as the RFBM, enclave code executing on another thread is free to modify the SSA contents before `ERESUME` is called.

The correct sequencing of actions in the **ERESUME** implementation prevents a malicious application from using an enclave to modify registers associated with extended architectural features that are not declared in **XFRM**. This would break the system software’s ability to provide thread-level execution context isolation.

## 5.5 EPC Page Eviction

Modern OS kernels take advantage of address translation (§ 2.5) to implement page swapping, also referred to as paging (§ 2.5). In a nutshell, paging allows the OS kernel to over-commit the computer’s DRAM by evicting rarely used memory pages to a slower storage medium called the disk.

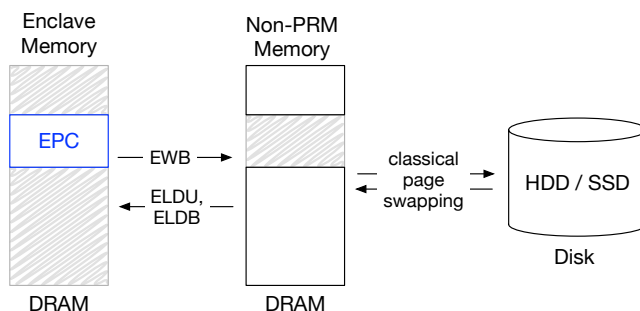
Paging is a key contributor to utilizing a computer’s resources effectively. For example, a desktop system whose user runs multiple programs concurrently can evict memory pages allocated to inactive applications without a significant degradation in user experience.

Unfortunately, the OS cannot be allowed to evict an enclave’s EPC pages via the same methods that are used to implement page swapping for DRAM memory outside the PRM range. In the SGX threat model, enclaves do not trust the system software, so the SGX design offers an EPC page eviction method that can defend against a malicious OS that attempts any of the active address translation attacks described in § 3.7.

The price of the security afforded by SGX is that an OS kernel that supports evicting EPC pages must use a modified page swapping implementation that interacts with the SGX mechanisms. Enclave authors can mostly ignore EPC evictions, similarly to how today’s application developers can ignore the OS kernel’s paging implementation.

As illustrated in Figure 5.10, SGX supports evicting EPC pages to DRAM pages outside the PRM range. The system software is expected to use its existing page swapping implementation to evict the contents of these pages out of DRAM and onto a disk.

SGX’s eviction feature revolves around the **EWB** instruction, described in detail in § 5.5.4. Essentially, **EWB** evicts an EPC page into a



**Figure 5.10:** SGX offers a method for the OS to evict EPC pages into non-PRM DRAM. The OS can then use its standard paging feature to evict the pages out of DRAM.

DRAM page outside the EPC and marks the EPC page as available, by zeroing the VALID field in the page’s EPCM entry.

The SGX design relies on symmetric key cryptography (§ 3.1.1) to guarantee the confidentiality and integrity of the evicted EPC pages, and on nonces (§ 3.1.4) to guarantee the freshness of the pages brought back into the EPC. These nonces are stored in Version Arrays (VAs), covered in § 5.5.2, which are EPC pages dedicated to nonce storage.

Before an EPC page is evicted and freed up for use by other enclaves, the SGX implementation must ensure that no TLB has address translations associated with the evicted page, in order to avoid the TLB-based address translation attack described in § 3.7.4.

As explained in § 5.1.1, SGX leaves the system software in charge of managing the EPC. It naturally follows that the SGX instructions described in this section, which are used to implement EPC paging, are only available to system software, which runs at ring 0 (§ 2.3).

In today’s software stacks (§ 2.3), only the OS kernel implements page swapping in order to support the over-committing of DRAM. The hypervisor is only used to partition the computer’s physical resources between operating systems. Therefore, this section is written with the expectation that the OS kernel will also take on the responsibility of EPC page swapping. For simplicity, we often use the term “OS kernel” instead of “system software”. The reader should be aware that the SGX design does not preclude a system where the hypervisor implements its

own EPC page swapping. Therefore, “OS kernel” should really be read as “the system software that performs EPC paging”.

### 5.5.1 Page Eviction and the TLBs

One of the least promoted accomplishments of SGX is that it does not add any security checks to the memory execution units (§ 2.9.4, § 2.10). Instead, SGX’s access control checks occur after an address translation (§ 2.5) is performed, right before the translation result is written into the TLBs (§ 2.11.5). This aspect is generally downplayed throughout the SDM, but it becomes visible when explaining SGX’s EPC page eviction mechanism.

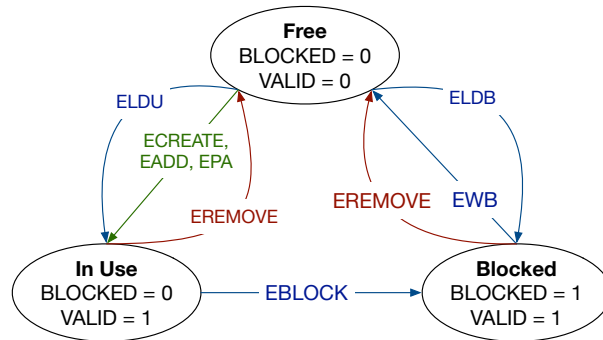
A full discussion of SGX’s memory access protections checks merits its own section, and is deferred to part II of this work. The EPC page eviction mechanisms can be explained using only two requirements from SGX’s security model. First, when a logical processor exits an enclave, either via `EEXIT` (§ 5.4.2) or via an `AEX` (§ 5.4.3), its TLBs are flushed. Second, when an EPC page is deallocated from an enclave, all logical processors executing that enclave’s code must be directed to exit the enclave. This is sufficient to guarantee the removal of any TLB entry targeting the deallocated EPC.

System software can cause a logical processor to exit an enclave by sending it an Inter-Processor Interrupt (IPI, § 2.12), which will trigger an `AEX` when received. Essentially, this is a very coarse-grained TLB shutdown.

SGX does not trust system software. Therefore, before marking an EPC page’s EPCM entry as free, the SGX implementation must ensure that the OS kernel has flushed all TLBs that may contain translations for the page. Furthermore, performing IPIs and TLB flushes for each page eviction would add a significant overhead to a paging implementation, so the SGX design allows a batch of pages to be evicted using a single IPI / TLB flush sequence.

The TLB flush verification logic relies on a 1-bit EPCM entry field called `BLOCKED`. As shown in Figure 5.11, the `VALID` and `BLOCKED` fields yield three possible EPC page states. A page is *free*

when both bits are zero, *in use* when VALID is zero and BLOCKED is one, and *blocked* when both bits are one.



**Figure 5.11:** The VALID and BLOCKED bits in an EPC page’s EPCM entry can be in one of three states. **EADD** and its siblings allocate new EPC pages. **EREMOVE** permanently deallocates an EPC page. **EBLOCK** blocks an EPC page so it can be evicted using **EWB**. **ELDB** and **ELDU** load an evicted page back into the EPC.

Blocked pages are not considered accessible to enclaves. If an address translation results in a blocked EPC page, the SGX implementation causes the translation to result in a Page Fault (#PF, § 2.8.2). This guarantees that once a page is blocked, the CPU will not create any new TLB entries pointing to it.

Furthermore, every SGX instruction makes sure that the EPC pages on which it operates are not blocked. For example, **EENTER** ensures that the TCS it is given is not blocked, that its enclave’s SECS is not blocked, and that every page in the current SSA is not blocked.

In order to evict a batch of EPC pages, the OS kernel must first issue **EBLOCK** instructions targeting them. The OS is also expected to remove the EPC page’s mapping from page tables, but is not trusted to do so.

After all desired pages have been blocked, the OS kernel must execute an **ETRACK** instruction, which directs the SGX implementation to keep track of which logical processors have had their TLBs flushed. **ETRACK** requires the virtual address of an enclave’s SECS (§ 5.1.3). If the OS wishes to evict a batch of EPC pages belonging to multiple enclaves, it must issue an **ETRACK** for each enclave.

Following the **ETRACK** instructions, the OS kernel must induce enclave exits on all logical processors that are executing code inside the enclaves that have been **ETRACKed**. The SGX design expects that the OS will use IPIs to cause AEXs in the logical processors whose TLBs must be flushed.

The EPC page eviction process is completed when the OS executes an **EWB** instruction for each EPC page to be evicted. This instruction, which will be fully described in § 5.5.4, writes an encrypted version of the EPC page to be evicted into DRAM, and then frees the page by clearing the **VALID** and **BLOCKED** bits in its EPCM entry. Before carrying out its tasks, **EWB** ensures that the EPC page that it targets has been blocked, and checks the state set up by **ETRACK** to make sure that all relevant TLBs have been flushed.

An evicted page can be loaded back into the EPC via the **ELDU** and **ELDB** instructions. Both instructions start up with a free EPC page and a DRAM page that has the evicted contents of an EPC page, decrypt the DRAM page's contents into the EPC page, and restore the corresponding EPCM entry. The only difference between **ELDU** and **ELDB** is that the latter sets the **BLOCKED** bit in the page's EPCM entry, whereas the former leaves it cleared.

**ELDU** and **ELDB** resemble **ECREATE** and **EADD**, in the sense that they populate a free EPC page. Since the page that they operate on was free, the SGX security model predicates that no TLB entries can possibly target it. Therefore, these instructions do not require a mechanism similar to **EBLOCK** or **ETRACK**.

### 5.5.2 The Version Array (VA)

When **EWB** evicts the contents of an EPC, it creates an 8-byte nonce (§ 3.1.4) that Intel's documentation calls a *page version*. SGX's freshness guarantees are built on the assumption that nonces are stored securely, so **EWB** stores the nonce that it creates inside a *Version Array* (VA).

Version Arrays are EPC pages that are dedicated to storing nonces generated by **EWB**. Each VA is divided into slots, and each slot is exactly large enough to store one nonce. Given that the size of an

EPC page is 4KB, and each nonce occupies 8 bytes, it follows that each VA has 512 slots.

VA pages are allocated using the **EPA** instruction, which takes in the virtual address of a free EPC page, and turns it into a Version Array with empty slots. VA pages are identified by the **PT\_VA** type in their EPCM entries. Like SECS pages, VA pages have the **ENCLAVEADDRESS** fields in their EPCM entries set to zero, and cannot be accessed directly by any software, including enclaves.

Unlike the other page types discussed so far, VA pages are not associated with any enclave. This means they can be deallocated via **EREMOVE** without any restriction. However, freeing up a VA page whose slots are in use effectively discards the nonces in those slots, which results in losing the ability to load the corresponding evicted pages back into the EPC. Therefore, it is unlikely that a correct OS implementation will ever call **EREMOVE** on a VA with non-free slots.

According to the pseudo-code for **EPA** and **EWB** in the SDM, SGX uses the zero value to represent the free slots in a VA, implying that all generated nonces have to be non-zero. This also means that **EPA** initializes a VA simply by zeroing the underlying EPC page. However, since software cannot access a VA's contents, neither the use of a special value, nor the value itself is architectural.

### 5.5.3 Enclave IDs

The **EWB** and **ELDU** / **ELDB** instructions use an *enclave ID* (EID) to identify the enclave that owns an evicted page. The EID has the same purpose as the **ENCLAVESECS** (§ 5.1.2) field in an EPCM entry, which is also used to identify the enclave that owns an EPC page. This section explains the need for having two values represent the same concept by comparing the two values and their uses.

The SDM states that **ENCLAVESECS** field in an EPCM entry is used to identify the SECS of the enclave owning the associated EPC page, but stops short of describing its format. In theory, the **ENCLAVESECS** field can change its representation between SGX implementations since SGX instructions never expose its value to software.



However, we will later argue that the most plausible representation of the ENCLAVESECS field is the physical address of the enclave's SECS. Therefore, the ENCLAVESECS value associated with a given enclave will change if the enclave's SECS is evicted from the EPC and loaded back at a different location. It follows that the ENCLAVESECS value is only suitable for identifying an enclave while its SECS remains in the EPC.

According to the SDM, the EID field is a 64-bit field stored in an enclave's SECS. ECREATE's pseudocode in the SDM reveals that an enclave's ID is generated when the SECS is allocated, by atomically incrementing a global counter. Assuming that the counter does not roll over<sup>4</sup>, this process guarantees that every enclave created during a power cycle has a unique EID.

Although the SDM does not specifically guarantee this, the EID field in an enclave's SECS does not appear to be modified by any instruction. This makes the EID's value suitable for identifying an enclave throughout its lifetime, even across evictions of its SECS page from the EPC.

#### 5.5.4 Evicting an EPC Page

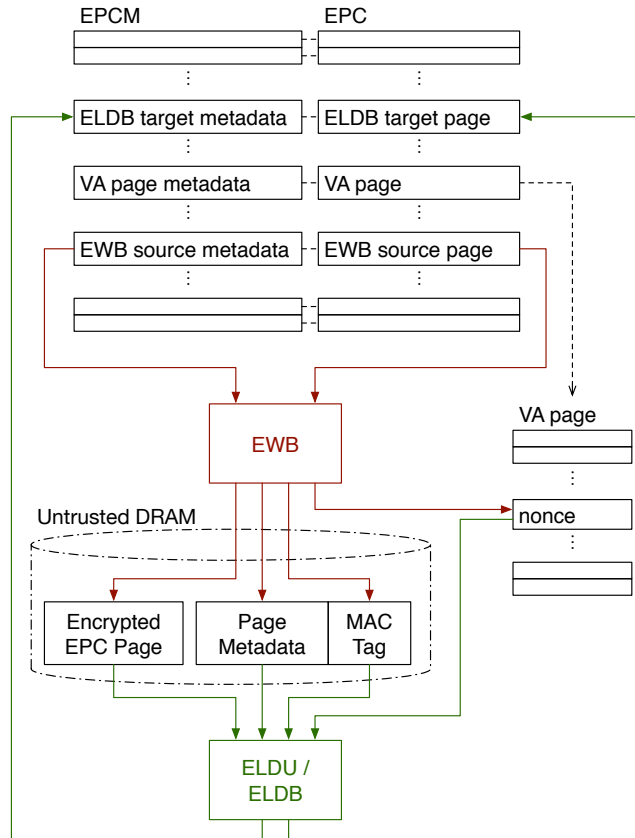
The system software evicts an EPC page using the EWB instruction, which produces all data needed to restore the evicted page at a later time via the ELDU instruction, as shown in Figure 5.12.

EWB's output consists of an encrypted version of the evicted EPC page's contents, a subset of the fields in the EPCM entry corresponding to the page, the nonce discussed in § 5.5.2, and a message authentication code (MAC, § 3.1.3) tag. With the exception of the nonce, EWB writes its output in DRAM outside the PRM area, so the system software can choose to further evict it to disk.

The EPC page contents is encrypted, to protect the confidentiality of the enclave's data while the page is stored in the untrusted DRAM outside the PRM range. Without the use of encryption, the system software could learn the contents of an EPC page by evicting it from the EPC.

---

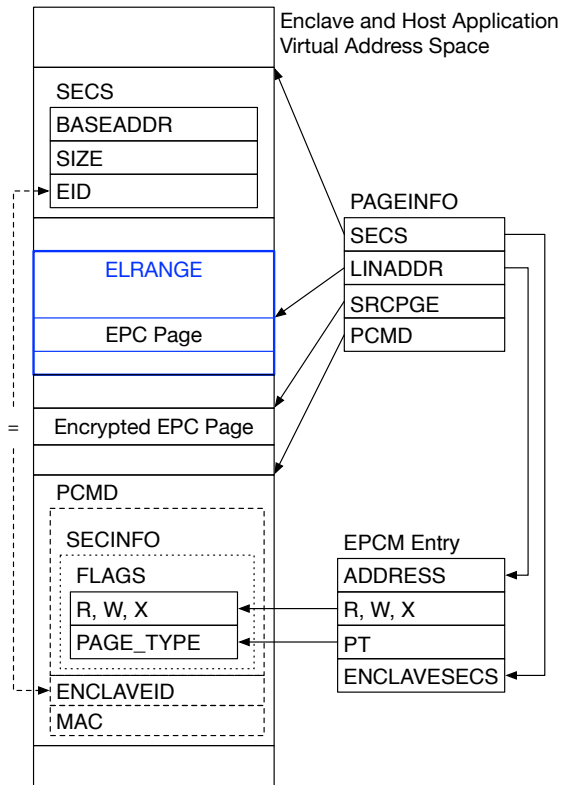
<sup>4</sup>A 64-bit counter incremented at 4GHz rolls over in slightly more than 136 years.



**Figure 5.12:** The EWB instruction outputs the encrypted contents of the evicted EPC page, a subset of the fields in the page’s EPCM entry, a MAC tag, and a nonce. All this information is used by the ELDB or ELDU instruction to load the evicted page back into the EPC, with confidentiality, integrity and freshness guarantees.

The page metadata is stored in a *Page Information* (PAGE-INFO) structure, illustrated in Figure 5.13. This structure is similar to the PAGEINFO structure described in § 5.3.2 and depicted in Figure 5.5, except that the SECINFO field has been replaced by a PCMD field, which contains the virtual address of a *Page Crypto Metadata* (PCMD) structure.

The LINADDR field in the PAGEINFO structure is used to store the ADDRESS field in the EPCM entry, which indicates the virtual



**Figure 5.13:** The PAGEINFO structure used by the EWB and ELDU / ELDB instructions.

address intended for accessing the page. The PCMD structure embeds the *Security Information* (SECINFO) described in § 5.3.2, which is used to store the page type (PT) and the access permission flags (R, W, X) in the EPCM entry. The PCMD structure also stores the enclave's ID (EID, § 5.5.3). These fields are later used by ELDU or ELDB to populate the EPCM entry for the EPC page that is reloaded.

The metadata described above is stored unencrypted, so the OS has the option of using the information inside as-is for its own bookkeeping. This has no negative impact on security, because the metadata is not confidential. In fact, with the exception of the enclave ID, all metadata fields are specified by the system software when ECREATE is called. The

enclave ID is only useful for identifying the enclave that the EPC page belongs to, and the system software already has this information as well.

Asides from the metadata described above, the PCMD structure also stores the MAC tag generated by EWB. The MAC tag covers the authenticity of the EPC page contents, the metadata, and the nonce. The MAC tag is checked by ELDU and ELDB, which will only load an evicted page back into the EPC if the MAC verification confirms the authenticity of the page data, metadata, and nonce. This security check protects against the page swapping attacks described in § 3.7.3.

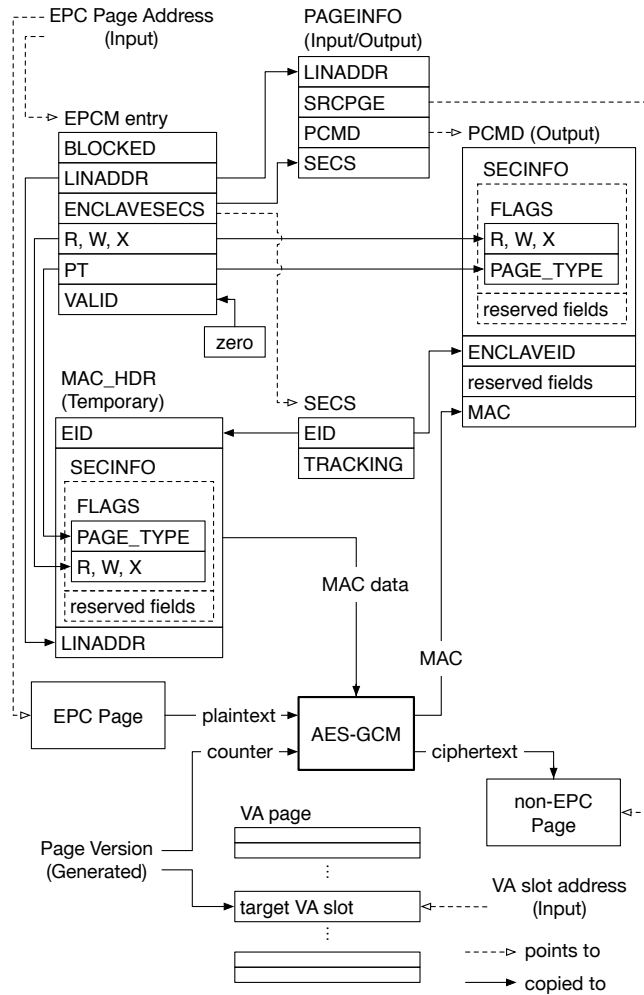
Similarly to EREMOVE, EWB will only evict the EPC page holding an enclave's SECS if there is no other EPCM entry whose ENCLAVESECS field references the SECS. At the same time, as an optimization, the SGX implementation does not perform ETRACK-related checks when evicting a SECS. This is safe because a SECS is only evicted if the EPC has no pages belonging to the SECS' enclave, which implies that there isn't any TCS belonging to the enclave in the EPC, so no processor can be executing enclave code.

The pages holding Version Arrays can be evicted, just like any other EPC page. VA pages are never accessible by software, so they can't have any TLB entries pointing to them. Therefore, EWB evicts VA pages without performing any ETRACK-related checks. The ability to evict VA pages has profound implications that will be discussed in § 5.5.6.

EWB's data flow, shown in detail in Figure 5.14, has an aspect that can be confusing to OS developers. The instruction reads the virtual address of the EPC page to be evicted from a register (RBX) and writes it to the LINADDR field of the PAGEINFO structure that it is provided. The separate input (RBX) could have been removed by providing the EPC page's address in the LINADDR field.

### 5.5.5 Loading an Evicted Page Back into EPC

After an EPC page belonging to an enclave is evicted, any attempt to access the page from enclave code will result in a Page Fault (#PF, § 2.8.2). The #PF will cause the logical processor to exit enclave mode via AEX (§ 5.4.3), and then invoke the OS kernel's page fault handler.



**Figure 5.14:** The data flow of the EWB instruction that evicts an EPC page. The page’s content is encrypted in a non-EPC RAM page. A nonce is created and saved in an empty slot inside a VA page. The page’s EPCM metadata and a MAC are saved in a separate area in non-EPC memory.

Page faults receive special handling from the AEX process. While leaving the enclave, the AEX logic specifically checks if the hardware exception that triggered the AEX was  $\#PF$ . If that is the case, the AEX

implementation clears the least significant 12 bits of the CR2 register, which stores the virtual address whose translation caused a page fault.

In general, the OS kernel's page handler needs to be able to extract the virtual page number (VPN, § 2.5.1) from CR2, so that it knows which memory page needs to be loaded back into DRAM. The OS kernel may also be able to use the 12 least significant address bits, which are not part of the VPN, to better predict the application software's memory access patterns. However, unlike the bits that make up the VPN, the bottom 12 bits are not absolutely necessary for the fault handler to carry out its job. Therefore, SGX's AEX implementation clears these 12 bits, in order to limit the amount of information that is learned by the page fault handler.

When the OS page fault handler examines the address in the CR2 register and determines that the faulting address is inside the EPC, it is generally expected to use the `ELDU` or `ELDB` instruction to load the evicted page back into the EPC. If the outputs of `EWB` have been evicted from DRAM to a slower storage medium, the OS kernel will have to read the outputs back into DRAM before invoking `ELDU` / `ELDB`.

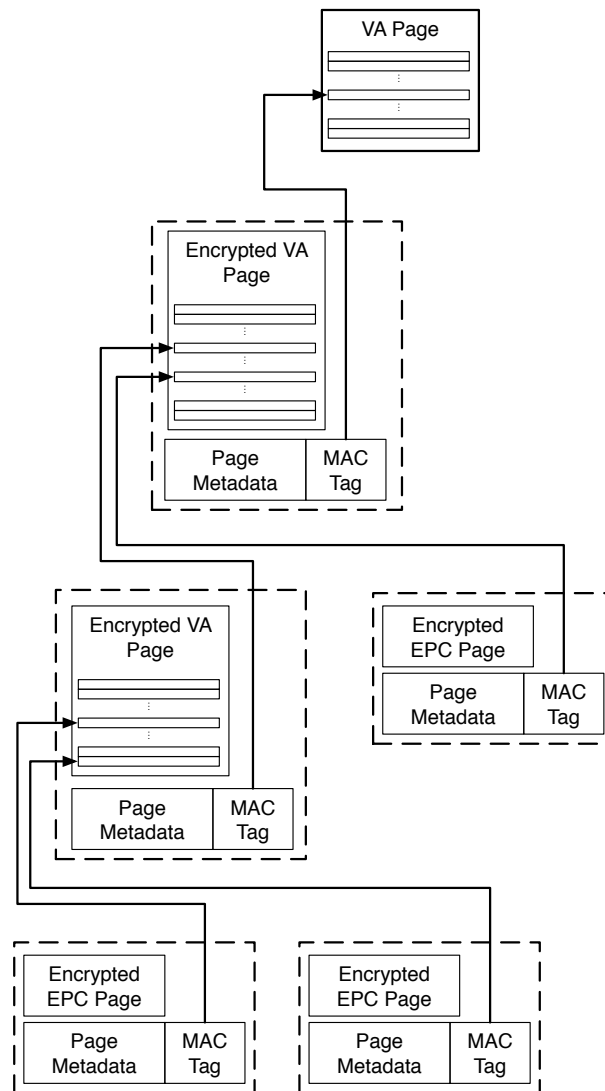
`ELDU` and `ELDB` verify the MAC tag produced by `EWB`, described in § 5.5.4. This prevents the OS kernel from performing the page swapping-based active address translation attack described in § 3.7.3.

### 5.5.6 Eviction Trees

The SGX design allows VA pages to be evicted from the EPC, just like enclave pages. When a VA page is evicted from EPC, all nonces stored by the VA slots become inaccessible to the processor. Therefore, the evicted pages associated with these nonces cannot be restored by `ELDB` until the OS loads the VA page back into the EPC.

In other words, an evicted page depends on the VA page storing its nonce, and cannot be loaded back into the EPC until the VA page is reloaded as well. The dependency graph created by this relationship is a forest of **eviction trees**. An eviction tree, shown in Figure 5.15, has enclave EPC pages as leaves, and VA pages as inner nodes. A page's parent is the VA page that holds its nonce. Since `EWB` always

outputs a nonce in a VA page, the root node of each eviction tree is always a VA page in the EPC.



**Figure 5.15:** A version tree formed by evicted VA pages and enclave EPC pages. The enclave pages are leaves, and the VA pages are inner nodes. The OS controls the tree’s shape, which impacts the performance of evictions, but not their correctness.

A straightforward inductive argument shows that when an OS wishes to load an evicted enclave page back into the EPC, it needs to load all VA pages on the path from the eviction tree's root to the leaf corresponding to the enclave page. Therefore, the number of page loads required to satisfy a page fault inside the EPC depends on the shape of the eviction tree that contains the page.

The SGX design leaves the OS in complete control of the shape of the eviction trees. This has no negative impact on security, as the tree shape only impacts the performance of the eviction scheme, and not its correctness.

## 5.6 SGX Enclave Measurement

SGX implements a software attestation scheme that follows the general principles outlined in § 3.3. For the purposes of this section, the most relevant principle is that a remote party authenticates an enclave based on its measurement, which is intended to identify the software that is executing inside the enclave. The remote party compares the enclave measurement reported by the trusted hardware with an expected measurement, and only proceeds if the two values match.

§ 5.3 explains that an SGX enclave is built using the `ECREATE` (§ 5.3.1), `EADD` (§ 5.3.2) and `EEXTEND` instructions. After the enclave is initialized via `EINIT` (§ 5.3.3), the instructions mentioned above cannot be used anymore. As the SGX measurement scheme follows the principles outlined in § 3.3.3, the measurement of an SGX enclave is obtained by computing a secure hash (§ 3.1.3) over the inputs to the `ECREATE`, `EADD` and `EEXTEND` instructions used to create the enclave and load the initial code and data into its memory. `EINIT` finalizes the hash that represents the enclave's measurement.

Along with the enclave's contents, the enclave author is expected to specify the sequence of instructions that should be used in order to create an enclave whose measurement will match the expected value used by the remote party in the software attestation process. The `.so` and `.dll` dynamically loaded library file formats, which are SGX's intended enclave delivery methods, already include informal specifica-



tions for loading algorithms. We expect the informal loading specifications to serve as the starting points for specifications that prescribe the exact sequences of SGX instructions that should be used to create enclaves from `.so` and `.dll` files.

As argued in § 3.3.3, an enclave’s measurement is computed using a secure hashing algorithm, so the system software can only build an enclave that matches an expected measurement by following the exact sequence of instructions specified by the enclave’s author.

The SGX design uses the 256-bit SHA-2 [Barker et al., 2015] secure hash function to compute its measurements. SHA-2 is a block hash function (§ 3.1.3) that operates on 64-byte blocks, uses a 32-byte internal state, and produces a 32-byte output. Each enclave’s measurement is stored in the MRENCLAVE field of the enclave’s SECS. The 32-byte field stores the internal state and final output of the 256-bit SHA-2 secure hash function.

### 5.6.1 Measuring ECREATE

The ECREATE instruction, described in § 5.3.1, first initializes the MRENCLAVE field in the newly created SECS using the 256-bit SHA-2 initialization algorithm, and then extends the hash with the 64-byte block depicted in Table 5.4.

**Table 5.4:** 64-byte block extended into MRENCLAVE by ECREATE.

| Offset | Size | Description                 |
|--------|------|-----------------------------|
| 0      | 8    | “ECREATE\0”                 |
| 8      | 8    | SECS.SSAFRAMESIZE (§ 5.2.5) |
| 16     | 8    | SECS.SIZE (§ 5.2.1)         |
| 32     | 8    | 32 zero (0) bytes           |

The enclave’s measurement does not include the BASEADDR field. The omission is intentional, as it allows the system software to load an enclave at any virtual address inside a host process that satisfies the ELRANGE restrictions (§ 5.2.1), without changing the enclave’s measurement. This feature can be combined with a compiler that generates position-independent enclave code to obtain relocatable enclaves.

The enclave's measurement includes the `SSAFRAMESIZE` field, which guarantees that the SSAs (§ 5.2.5) created by AEX and used by `EENTER` (§ 5.4.1) and `ERESUME` (§ 5.4.4) have the size that is expected by the enclave's author. Leaving this field out of an enclave's measurement would allow a malicious enclave loader to attempt to attack the enclave's security checks by specifying a bigger `SSAFRAMESIZE` than the enclave's author intended, which could cause the SSA contents written by an AEX to overwrite the enclave's code or data.

### 5.6.2 Measuring Enclave Attributes

The enclave's measurement does not include the enclave attributes (§ 5.2.2), which are specified in the `ATTRIBUTES` field in the SECS. Instead, it is included directly in the information that is covered by the attestation signature, which will be discussed in § 5.8.1.

The SGX software attestation definitely needs to cover the enclave attributes. For example, if `XFRM` (§ 5.2.2, § 5.2.5) would not be covered, a malicious enclave loader could attempt to subvert an enclave's security checks by setting `XFRM` to a value that enables architectural extensions that change the semantics of instructions used by the enclave, but still produces an `XSAVE` output that fits in `SSAFRAMESIZE`.

The special treatment applied to the `ATTRIBUTES` SECS field seems questionable from a security standpoint, as it adds extra complexity to the software attestation verifier, which translates into more opportunities for exploitable bugs. This decision also adds complexity to the SGX software attestation design, which is described in § 5.8.

The most likely reason why the SGX design decided to go this route, despite the concerns described above, is the wish to be able to use a single measurement to represent an enclave that can take advantage of some architectural extensions, but can also perform its task without them.

Consider, for example, an enclave that performs image processing using a library such as OpenCV, which has routines optimized for SSE and AVX, but also includes generic fallbacks for processors that do not have these features. The enclave's author will likely wish to allow an enclave loader to set bits 1 (SSE) and 2 (AVX) to either true or false. If

ATTRIBUTES (and, by extension, XFRM) was a part of the enclave’s measurement, the enclave author would have to specify that the enclave has 4 valid measurements. In general, allowing  $n$  architectural extensions to be used independently will result in  $2^n$  valid measurements.

### 5.6.3 Measuring EADD

The *EADD* instruction, described in § 5.3.2, extends the SHA-2 hash in MRENCLAVE with the 64-byte block shown in Table 5.5.

**Table 5.5:** 64-byte block extended into MRENCLAVE by *EADD*. The ENCLAVE-OFFSET is computed by subtracting the BASEADDR in the enclave’s SECS from the LINADDR field in the PAGEINFO structure.

| Offset | Size | Description              |
|--------|------|--------------------------|
| 0      | 8    | “EADD\0\0\0\0”           |
| 8      | 8    | ENCLAVEOFFSET            |
| 16     | 48   | SECINFO (first 48 bytes) |

The address included in the measurement is the address where the *EADD*ed page is expected to be mapped in the enclave’s virtual address space. This ensures that the system software sets up the enclave’s virtual memory layout according to the enclave author’s specifications. If a malicious enclave loader attempts to set up the enclave’s layout incorrectly, perhaps in order to mount an active address translation attack (§ 3.7.2), the loaded enclave’s measurement will differ from the measurement expected by the enclave’s author.

The virtual address of the newly created page is measured relative to the start of the enclave’s ELRANGE. In other words, the value included in the measurement is LINADDR - BASEADDR. This makes the enclave’s measurement invariant to BASEADDR changes, which is desirable for relocatable enclaves. Measuring the relative addresses still preserves all information about the memory layout inside ELRANGE, and therefore has no negative security impact.

*EADD* also measures the first 48 bytes of the SECINFO structure (§ 5.3.2) provided to *EADD*, which contain the page type (PT) and access permissions (R, W, X) field values used to initialize the

page's EPCM entry. By the same argument as above, including these values in the measurement guarantees that the memory layout built by the system software loading the enclave matches the specifications of the enclave author.

The EPCM field values mentioned above take up less than one byte in the SECINFO structure, and the rest of the bytes are reserved and expected to be initialized to zero. This leaves plenty of expansion room for future SGX features.

The most notable omission from Table 5.5 is the data used to initialize the newly created EPC page. Therefore, the measurement data contributed by EADD guarantees that the enclave's memory layout will have pages allocated with prescribed access permissions at the desired virtual addresses. However, the measurements don't cover the code or data loaded in these pages.

For example, EADD's measurement data guarantees that an enclave's memory layout consists of three executable pages followed by five writable data pages, but it does not guarantee that any of the code pages contains the code supplied by the enclave's author.

#### 5.6.4 Measuring EEXTEND

The EEXTEND instruction exists solely for the reason of measuring data loaded inside the enclave's EPC pages. The instruction reads in a virtual address, and extends the enclave's measurement hash with the five 64-byte blocks in Table 5.6, which effectively guarantee the contents of a 256-byte chunk of data in the enclave's memory.

Before examining the details of EEXTEND, we note that SGX's security guarantees only hold when the contents of the enclave's key pages is measured. For example, EENTER (§ 5.4.1) is only guaranteed to perform controlled jumps inside an enclave's code if the contents of all Thread Control Structure (TCS, § 5.2.4) pages are measured. Otherwise, a malicious enclave loader can change the OENTRY field (§ 5.2.4, § 5.4.1) in a TCS while building the enclave, and then a malicious OS can use the TCS to perform an arbitrary jump inside enclave code. By the same argument, the entire body of the enclave's code should be

**Table 5.6:** 64-byte blocks extended into MRENCLAVE by **EEXTEND**. The ENCLAVEOFFSET is computed by subtracting the BASEADDR in the enclave’s SECS from the LINADDR field in the PAGEINFO structure.

| Offset | Size | Description                  |
|--------|------|------------------------------|
| 0      | 8    | “EEXTEND\0”                  |
| 8      | 8    | ENCLAVEOFFSET                |
| 16     | 48   | 48 zero (0) bytes            |
| 64     | 64   | bytes 0 - 64 in the chunk    |
| 128    | 64   | bytes 64 - 128 in the chunk  |
| 192    | 64   | bytes 128 - 192 in the chunk |
| 256    | 64   | bytes 192 - 256 in the chunk |

measured by **EEXTEND**. Any code fragment that is not measured can be replaced by a malicious enclave loader.

Given these pitfalls, it is surprising that the SGX design opted to decouple the virtual address space layout measurements done by **EADD** from the memory content measurements done by **EEXTEND**.

At a first pass, it appears that the decoupling only has one benefit, which is the ability to load unmeasured user input into an enclave while it is being built. However, this benefit only translates into a small performance improvement, because enclaves can alternatively be designed to copy the user input from untrusted DRAM after being initialized. At the same time, the decoupling opens up the possibility of relying on an enclave that provides no meaningful security guarantees, due to not measuring all important data via **EEXTEND** calls.

However, the real reason behind the **EADD** / **EEXTEND** separation is hinted at by the **EINIT** pseudo-code in the SDM, which states that the instruction opens an interrupt (§ 2.12) window while it performs a computationally intensive RSA signature check. If an interrupt occurs during the check, **EINIT** fails with an error code, and the interrupt is serviced. This very unusual approach for a processor instruction suggests that the SGX implementation was constrained in respect to how much latency its instructions were allowed to add to the interrupt handling process.

In light of the concerns above, it is reasonable to conclude that `EEXTEND` was introduced because measuring an entire page using 256-bit SHA-2 is quite time-consuming, and doing it in `EADD` would have caused the instruction to exceed SGX's latency budget. The need to hit a certain latency goal is a reasonable explanation for the seemingly arbitrary 256-byte chunk size.

The `EADD` / `EEXTEND` separation will not cause security issues if enclaves are authored using the same tools that build today's dynamically loaded modules, which appears to be the workflow targeted by the SGX design. In this workflow, the tools that build enclaves can easily identify the enclave data that needs to be measured.

It is correct and meaningful, from a security perspective, to have the message blocks provided by `EEXTEND` to the hash function include the address of the 256-byte chunk, in addition to the contents of the data. If the address were not included, a malicious enclave loader could mount the memory mapping attack described in § 3.7.2 and illustrated in Figure 3.23.

More specifically, the malicious loader would `EADD` the `errorOut` page contents at the virtual address intended for `disclose`, `EADD` the `disclose` page contents at the virtual address intended for `errorOut`, and then `EEXTEND` the pages in the wrong order. If `EEXTEND` would not include the address of the data chunk that is measured, the steps above would yield the same measurement as the correctly constructed enclave.

The last aspect of `EEXTEND` worth analyzing is its support for relocating enclaves. Similarly to `EADD`, the virtual address measured by `EEXTEND` is relative to the enclave's `BASEADDR`. Furthermore, the only SGX structure whose content is expected to be measured by `EEXTEND` is the TCS. The SGX design has carefully used relative addresses for all TCS fields that represent enclave addresses, which are `OENTRY`, `OFSBASGX` and `OGSBASGX`.

### 5.6.5 Measuring `EINIT`

The `EINIT` instruction (§ 5.3.3) concludes the enclave building process. After `EINIT` is successfully invoked on an enclave, the enclave's contents are “sealed”, meaning that the system software cannot use the `EADD`

instruction to load code and data into the enclave, and cannot use the `EEXTEND` instruction to update the enclave's measurement.

`EINIT` uses the SHA-2 finalization algorithm (§ 3.1.3) on the `MRENCLAVE` field of the enclave's SECS. After `EINIT`, the field no longer stores the intermediate state of the SHA-2 algorithm, and instead stores the final output of the secure hash function. This value remains constant after `EINIT` completes, and is included in the attestation signature produced by the SGX software attestation process.

## 5.7 SGX Enclave Versioning Support

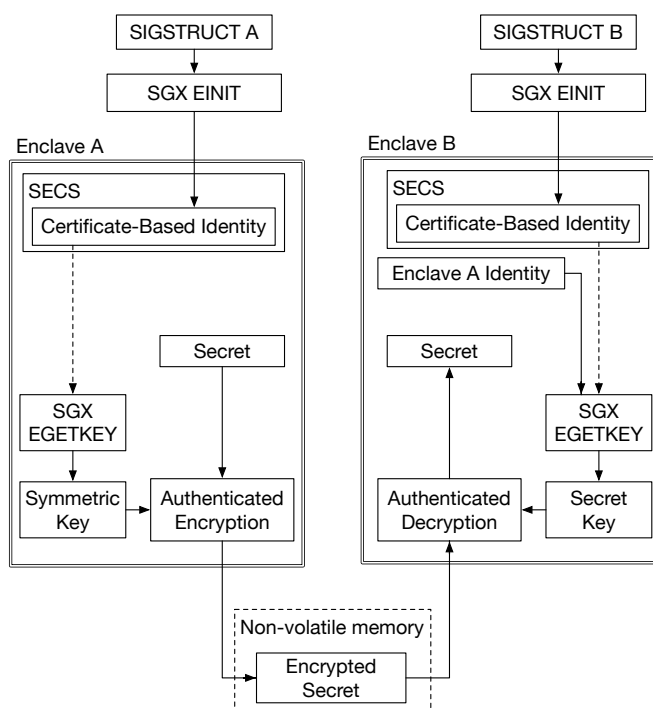
The software attestation model (§ 3.3) introduced by the Trusted Platform Module (§ 4.4) relies on a measurement (§ 5.6), which is essentially a content hash, to identify the software inside a container. The downside of using content hashes for identity is that there is no relation between the identities of containers that hold different versions of the same software.

In practice, it is highly desirable for systems based on secure containers to handle software updates without having access to the remote party in the initial software attestation process. This entails having the ability to migrate secrets between the container that has the old version of the software and the container that has the updated version. This requirement translates into a need for a separate identity system that can recognize the relationship between two versions of the same software.

SGX supports the migration of secrets between enclaves that represent different versions of the same software, as shown in Figure 5.16.

The secret migration feature relies on a one-level certificate hierarchy (§ 3.2.1), where each enclave author is a Certificate Authority, and each enclave receives a certificate from its author. These certificates must be formatted as Signature Structures (`SIGSTRUCT`), which are described in § 5.7.1. The information in these certificates is the basis for an enclave identity scheme, presented in § 5.7.2, which can recognize the relationship between different versions of the same software.

The `EINIT` instruction (§ 5.3.3) examines the target enclave's certificate and uses the information in it to populate the SECS (§ 5.1.3)



**Figure 5.16:** SGX has a certificate-based enclave identity scheme, which can be used to migrate secrets between enclaves that contain different versions of the same software module. Here, enclave A’s secrets are migrated to enclave B.

fields that describe the enclave’s certificate-based identity. This process is summarized in § 5.7.4.

Last, the actual secret migration process is based on the key derivation service implemented by the **EGETKEY** instruction, which is described in § 5.7.5. The sending enclave uses the **EGETKEY** instruction to obtain a symmetric key (§ 3.1.1) based on its identity, encrypts its secrets with the key, and hands off the encrypted secrets to the untrusted system software. The receiving enclave passes the sending enclave’s identity to **EGETKEY**, obtains the same symmetric key as above, and uses the key to decrypt the secrets received from system software.

The symmetric key obtained from **EGETKEY** can be used in conjunction with cryptographic primitives that protect the confidentiality (§ 3.1.2) and integrity (§ 3.1.3) of an enclave’s secrets while they



are migrated to another enclave by the untrusted system software. However, symmetric keys alone cannot be used to provide freshness guarantees (§ 3.1), so secret migration is subject to replay attacks. This is acceptable when the secrets being migrated are immutable, such as when the secrets are encryption keys obtained via software attestation.

### 5.7.1 Enclave Certificates

The SGX design requires each enclave to have a certificate issued by its author. This requirement is enforced by `EINIT` (§ 5.3.3), which refuses to operate on enclaves without valid certificates.

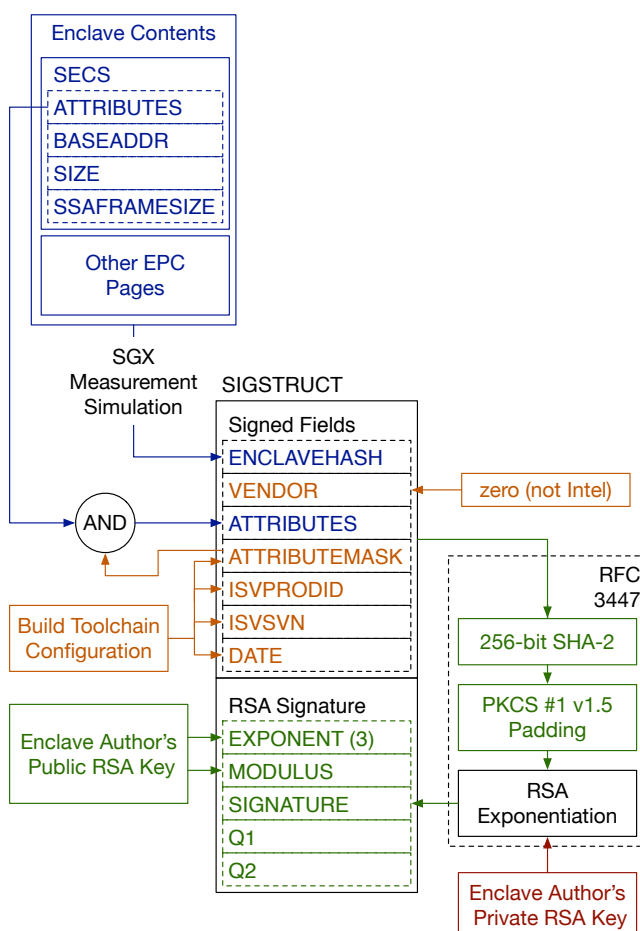
The SGX implementation consumes certificates formatted as *Signature Structures* (SIGSTRUCT), which are intended to be generated by an enclave building toolchain, as shown in Figure 5.17.

A SIGSTRUCT certificate consists of metadata fields, the most interesting of which are presented in Table 5.7, and an RSA signature that guarantees the authenticity of the metadata, formatted as shown in Table 5.8. The semantics of the fields will be revealed in the following sections.

**Table 5.7:** A subset of the metadata fields in a SIGSTRUCT enclave certificate.

| Field         | Bytes | Description   |
|---------------|-------|---|
| ENCLAVEHASH   | 32    | Must equal the enclave’s measurement (§ 5.6).         |
| ISVPRODID     | 32    | Differentiates modules signed by the same public key. |
| ISVSVN        | 32    | Differentiates versions of the same module.           |
| VENDOR        | 4     | Differentiates Intel enclaves.                        |
| ATTRIBUTES    | 16    | Constrains the enclave’s attributes.                  |
| ATTRIBUTEMASK | 16    | Constrains the enclave’s attributes.                  |

The enclave certificates must be signed by RSA signatures (§ 3.1.3) that follow the method described in RFC 3447 [Jonsson and Kaliski, 2003], using 256-bit SHA-2 [Barker et al., 2015] as the hash function



**Figure 5.17:** An enclave's Signature Structure (SIGSTRUCT) is intended to be generated by an enclave building toolchain that has access to the enclave author's private RSA key.

that reduces the input size, and the padding method described in PKCS #1 v1.5 [Kaliski, 1998], which is illustrated in Figure 3.14.

The SGX implementation only supports 3072-bit RSA keys whose public exponent is 3. The key size is likely chosen to meet FIPS' recommendation [Barker et al., 2012], which makes SGX eligible for use in U.S. government applications. The public exponent 3 affords a simplified signature verification algorithm, which is discussed in § II.2.5.

**Table 5.8:** The format of the RSA signature used in a SIGSTRUCT enclave certificate.

| Field     | Bytes | Description   |
|-----------|-------|---|
| MODULUS   | 384   | RSA key modulus                                       |
| EXPONENT  | 4     | RSA key public exponent                               |
| SIGNATURE | 384   | RSA signature (See § II.2.5)                          |
| Q1        | 384   | Simplifies RSA signature verification. (See § II.2.5) |
| Q2        | 384   | Simplifies RSA signature verification. (See § II.2.5) |

The simplified algorithm also requires the fields Q1 and Q2 in the RSA signature, which are also described in § II.2.5.

### 5.7.2 Certificate-Based Enclave Identity

An enclave’s identity is determined by three fields in its certificate (§ 5.7.1): the modulus of the RSA key used to sign the certificate (MODULUS), the enclave’s product ID (ISVPRODID) and the security version number (ISVSVN).

The public RSA key used to issue a certificate identifies the enclave’s author. All RSA keys used to issue enclave certificates must have the public exponent set to 3, so they are only differentiated by their modulus. SGX does not use the entire modulus of a key, but rather a 256-bit SHA-2 hash of the modulus. This is called a *signer measurement* (MRSIGNER), to parallel the name of *enclave measurement* (MRENCLAVE) for the SHA-2 hash that identifies an enclave’s contents.

The SGX implementation relies on a hard-coded MRSIGNER value to recognize certificates issued by Intel. Enclaves that have an Intel-issued certificate can receive additional privileges, which are discussed in § 5.8.

An enclave author can use the same RSA key to issue certificates for enclaves that represent different software modules. Each module is identified by a unique Product ID (ISVPRODID) value. Conversely, all enclaves whose certificates have the same ISVPRODID and are issued

by the same RSA key (and therefore have the same MRENCLAVE) are assumed to represent different versions of the same software module. Enclaves whose certificates are signed by different keys are always assumed to contain different software modules.

Enclaves that represent different versions of a module can have different security version numbers (SVN). The SGX design disallows the migration of secrets from an enclave with a higher SVN to an enclave with a lower SVN. This restriction is intended to assist with the distribution of security patches, as follows.

If a security vulnerability is discovered in an enclave, the author can release a fixed version with a higher SVN. As users upgrade, SGX will facilitate the migration of secrets from the vulnerable version of the enclave to the fixed version. Once a user's secrets have migrated, the SVN restrictions in SGX will deflect any attack based on building the vulnerable enclave version and using it to read the migrated secrets.

Software upgrades that add functionality should not be accompanied by an SVN increase, as SGX allows secrets to be migrated freely between enclaves with matching SVN values. As explained above, a software module's SVN should only be incremented when a security vulnerability is found. SIGSTRUCT only allocates 2 bytes to the ISVSVN field, which translates to 65,536 possible SVN values. This space can be exhausted if a large team (incorrectly) sets up a continuous build system to allocate a new SVN for every software build that it produces, and each code change triggers a build.

### 5.7.3 CPU Security Version Numbers

The SGX implementation itself has a security version number (CPUSVN), which is used in the key derivation process implemented [McKeen et al., 2009] by `EGETKEY`, in addition to the enclave's identity information. CPUSVN is a 128-bit value that, according to the SDM, reflects the processor's microcode update version.

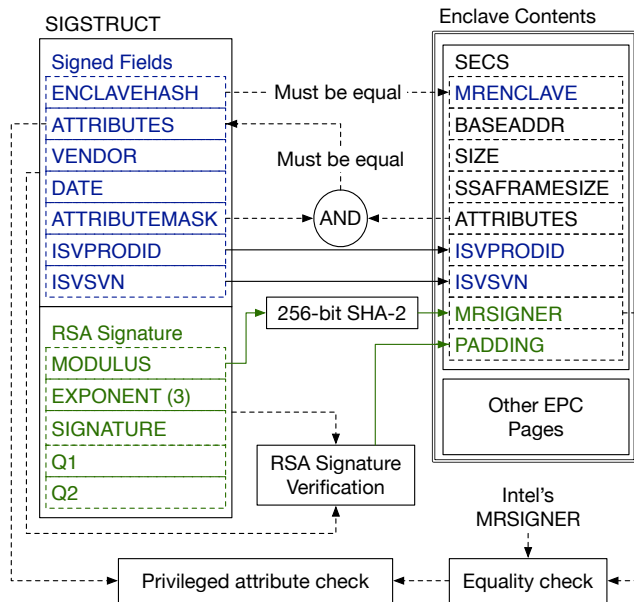
The SDM does not describe the structure of CPUSVN, but it states that comparing CPUSVN values using integer comparison is not meaningful, and that only some CPUSVN values are valid. Furthermore, CPUSVNs admit an ordering relationship that has the same

semantics as the ordering relationship between enclave SVNs. Specifically, an SGX implementation will consider all SGX implementations with lower SVNs to be compromised due to security vulnerabilities, and will not trust them.

An SGX patent [McKeen et al., 2009] discloses that CPUSVN is a concatenation of small integers representing the SVNs of the various components that make up SGX’s implementation. This structure is consistent with all statements made in the SDM.

#### 5.7.4 Establishing an Enclave’s Identity

When the EINIT (§ 5.3.3) instruction prepares an enclave for code execution, it also sets the SECS (§ 5.1.3) fields that make up the enclave’s certificate-based identity, as shown in Figure 5.18.



**Figure 5.18:** EINIT verifies the RSA signature in the enclave’s certificate. If the certificate is valid, the information in it is used to populate the SECS fields that make up the enclave’s certificate-based identity.

EINIT requires the virtual address of the SIGSTRUCT certificate issued to the enclave, and uses the information in the certificate to ini-

tialize the certificate-based identity information in the enclave's SECS. Before using the information in the certificate, `EINIT` first verifies its RSA signature. The `SIGSTRUCT` fields `Q1` and `Q2`, along with the RSA exponent 3, facilitate a simplified verification algorithm, which is discussed in § II.2.5.

If the `SIGSTRUCT` certificate is found to be properly signed, `EINIT` follows the steps discussed in the following few paragraphs to ensure that the certificate was issued to the enclave that is being initialized. Once the checks have completed, `EINIT` computes `MR-SIGNER`, the 256-bit SHA-2 hash of the `MODULUS` field in the `SIGSTRUCT`, and writes it into the enclave's SECS. `EINIT` also copies the `ISVPRODID` and `ISVSVN` fields from `SIGSTRUCT` into the enclave's SECS. As explained in § 5.7.2, these fields make up the enclave's certificate-based identity.

After verifying the RSA signature in `SIGSTRUCT`, `EINIT` copies the signature's padding into the `PADDING` field in the enclave's SECS. The PKCS #1 v1.5 padding scheme, outlined in Figure 3.14, does not involve randomness, so `PADDING` should have the same value for all enclaves.

`EINIT` performs a few checks to make sure that the enclave undergoing initialization was indeed authorized by the provided `SIGSTRUCT` certificate. The most obvious check involves making sure that the `MRENCLAVE` value in `SIGSTRUCT` equals the enclave's measurement, which is stored in the `MRENCLAVE` field in the enclave's SECS.

However, `MRENCLAVE` does not cover the enclave's attributes, which are stored in the `ATTRIBUTES` field of the SECS. As discussed in § 5.6.2, omitting `ATTRIBUTES` from `MRENCLAVE` facilitates writing enclaves that have optimized implementations that can use architectural extensions when present, and also have fallback implementations that work on CPUs without the extensions. Such enclaves can execute correctly when built with a variety of values in the `XFRM` (§ 5.2.2, § 5.2.5) attribute. At the same time, allowing system software to use arbitrary values in the `ATTRIBUTES` field would compromise SGX's security guarantees.

When an enclave uses software attestation (§ 3.3) to gain access to secrets, the `ATTRIBUTES` value used to build it is included in the SGX attestation signature (§ 5.8). This gives the remote party in the attestation process the opportunity to reject an enclave built with an undesirable `ATTRIBUTES` value. However, when secrets are obtained using the migration process facilitated by certificate-based identities, there is no remote party that can check the enclave's attributes.

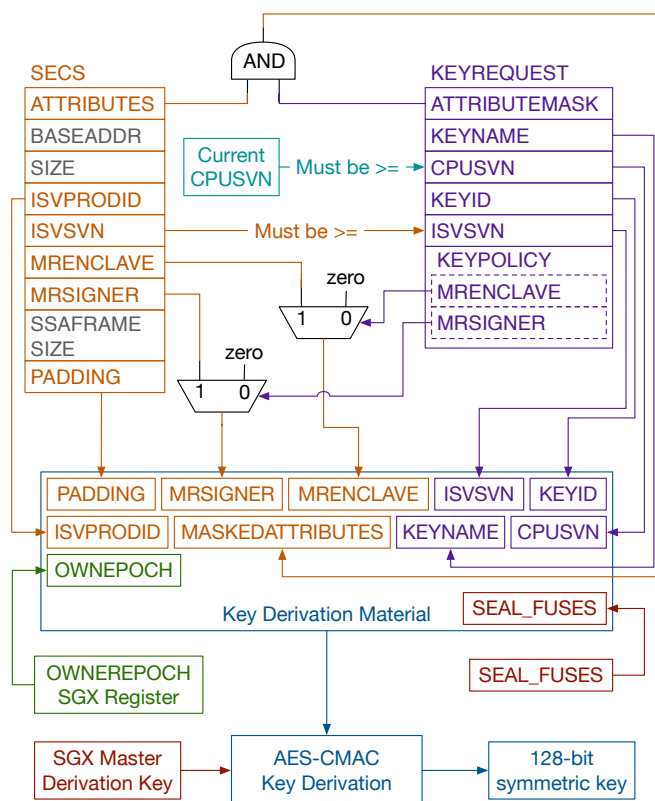
The SGX design solves this problem by having enclave authors convey the set of acceptable attribute values for an enclave in the `ATTRIBUTES` and `ATTRIBUTEMASK` fields of the `SIGSTRUCT` certificate issued for the enclave. `EINIT` will refuse to initialize an enclave using a `SIGSTRUCT` if the bitwise AND between the `ATTRIBUTES` field in the enclave's `SECS` and the `ATTRIBUTESMASK` field in the `SIGSTRUCT` does not equal the `SIGSTRUCT`'s `ATTRIBUTES` field. This check prevents enclaves with undesirable attributes from obtaining and potentially leaking secrets using the migration process.

Any enclave author can use `SIGSTRUCT` to request any of the bits in an enclave's `ATTRIBUTES` field to be zero. However, certain bits can only be set to one for enclaves that are signed by Intel. `EINIT` has a mask of restricted `ATTRIBUTES` bits, discussed in § 5.8. The `EINIT` implementation contains a hard-coded `MRSIGNER` value that is used to identify Intel's privileged enclaves, and only allows privileged enclaves to be built with an `ATTRIBUTES` value that matches any of the bits in the restricted mask. This check is essential to the security of the SGX software attestation process, which is described in § 5.8.

Last, `EINIT` also inspects the `VENDOR` field in `SIGSTRUCT`. The SDM description of the `VENDOR` field in the section dedicated to `SIGSTRUCT` suggests that the field is essentially used to distinguish between special enclaves signed by Intel, which use a `VENDOR` value of 0x8086, and everyone else's enclaves, which should use a `VENDOR` value of zero. However, the `EINIT` pseudocode seems to imply that the SGX implementation only checks that `VENDOR` is either zero or 0x8086.

### 5.7.5 Enclave Key Derivation

SGX's secret migration mechanism is based on the symmetric key derivation service that is offered to enclaves by the `EGETKEY` instruction, illustrated in Figure 5.19.



**Figure 5.19:** EGETKEY implements a key derivation service that is primarily used by SGX's secret migration feature. The key derivation material is drawn from the SECS of the calling enclave, the information in a Key Request structure, and secure storage inside the CPU's hardware.

The keys produced by `EGETKEY` are derived based on the identity information in the current enclave's SECS and on two secrets stored in secure hardware inside the SGX-enabled processor. One of the secrets is the input to a largely undocumented series of transformations that yields the symmetric key for the cryptographic primitive under-



lying the key derivation process. The other secret, referred to as the `CR_SEAL_FUSES` in the SDM, is one of the pieces of information used in the key derivation material.

The SDM does not specify the key derivation algorithm, but the SGX patents [McKeen et al., 2009, Johnson et al., 2010] disclose that the keys are derived using the method described in FIPS SP 800-108 [Chen, 2009] using AES-CMAC [Dworkin, 2005] as a Pseudo-Random Function (PRF). The same patents state that the secrets used for key derivation are stored in the CPU’s e-fuses, which is confirmed by the ISCA 2015 SGX tutorial [Int, 2015f].

This additional information implies that all `EGETKEY` invocations that use the same key derivation material will result in the same key, even across CPU power cycles. Furthermore, it is impossible for an adversary to obtain the key produced from a specific key derivation material without access to the secret stored in the CPU’s e-fuses. SGX’s key hierarchy is further described in § 5.8.2.

The following paragraphs discuss the pieces of data used in the key derivation material, which are selected by the Key Request (`KEYREQUEST`) structure shown in in Table 5.9,

**Table 5.9:** A subset of the fields in the `KEYREQUEST` structure.

| Field                      | Bytes | Description  |
|----------------------------|-------|--|
| <code>KEYNAME</code>       | 2     | The desired key type; secret migration uses Seal keys                            |
| <code>KEYPOLICY</code>     | 2     | The identity information ( <code>MRENCLAVE</code> and/or <code>MRSIGNER</code> ) |
| <code>ISVSVN</code>        | 2     | The enclave SVN used in derivation   |
| <code>CPUSVN</code>        | 16    | SGX implementation SVN used in derivation  |
| <code>ATTRIBUTEMASK</code> | 16    | Selects enclave attributes   |
| <code>KEYID</code>         | 32    | Random bytes   |

The `KEYNAME` field in `KEYREQUEST` always participates in the key generation material. It indicates the type of the key to be generated. While the SGX design defines a few key types, the secret migration

feature always uses Seal keys. The other key types are used by the SGX software attestation process, which will be outlined in § 5.8.

The `KEYPOLICY` field in `KEYREQUEST` has two flags that indicate if the `MRENCLAVE` and `MRSIGNER` fields in the enclave's SECS will be used for key derivation. Although the field admits 4 values, only two seem to make sense, as argued below.

Setting the `MRENCLAVE` flag in `KEYPOLICY` ties the derived key to the current enclave's measurement, which reflects its contents. No other enclave will be able to obtain the same key. This is useful when the derived key is used to encrypt enclave secrets so they can be stored by system software in non-volatile memory, and thus survive power cycles.

If the `MRSIGNER` flag in `KEYPOLICY` is set, the derived key is tied to the public RSA key that issued the enclave's certificate. Therefore, other enclaves issued by the same author may be able to obtain the same key, subject to the restrictions below. This is the only `KEYPOLICY` value that allows for secret migration.

It makes little sense to have no flag set in `KEYPOLICY`. In this case, the derived key has no useful security property, as it can be obtained by other enclaves that are completely unrelated to the enclave invoking `EGETKEY`. Conversely, setting both flags is redundant, as setting `MRENCLAVE` alone will cause the derived key to be tied to the current enclave, which is the strictest possible policy.

The `KEYREQUEST` structure specifies the enclave SVN (`ISVSVN`, § 5.7.2) and SGX implementation SVN (`CPUSVN`, § 5.7.3) that will be used in the key derivation process. However, `EGETKEY` will reject the derivation request and produce an error code if the desired enclave SVN is greater than the current enclave's SVN, or if the desired SGX implementation's SVN is greater than the current implementation's SVN.

The SVN restrictions prevent the migration of secrets from enclaves with higher SVNs to enclaves with lower SVNs, or from SGX implementations with higher SVNs to implementations with lower SVNs. § 5.7.2 argues that the SVN restrictions can reduce the impact of security vulnerabilities in enclaves and in SGX's implementation.

**EGETKEY** always uses the **ISVPRODID** value from the current enclave's **SECS** for key derivation. It follows that secrets can never flow between enclaves whose **SIGSTRUCT** certificates assign them different **Product IDs**.

Similarly, the key derivation material always includes the value of a 128-bit **Owner Epoch (OWNEREPOCH)** **SGX** configuration register. This register is intended to be set by the computer's firmware to a secret generated once and stored in non-volatile memory. Before the computer changes ownership, the old owner can clear the **OWNEREPOCH** from non-volatile memory, making it impossible for the new owner to decrypt any enclave secrets that may be left on the computer.

Due to the cryptographic properties of the key derivation process, outside observers cannot correlate keys derived using different **OWNEREPOCH** values. This makes it impossible for software developers to use the **EGETKEY**-derived keys described in this section to track a processor as it changes owners.

The **EGETKEY** derivation material also includes a 256-bit value supplied by the enclave, in the **KEYID** field. This makes it possible for an enclave to generate a collection of keys from **EGETKEY**, instead of a single key. The **SDM** states that **KEYID** should be populated with a random number, and is intended to help prevent key wear-out.

Last, the key derivation material includes the bitwise **AND** of the **ATTRIBUTES** (§ 5.2.2) field in the enclave's **SECS** and the **ATTRIBUTESMASK** field in the **KEYREQUEST** structure. The mask has the effect of removing some of the **ATTRIBUTES** bits from the key derivation material, making it possible to migrate secrets between enclaves with different attributes. § 5.6.2 and § 5.7.4 explain the need for this feature, as well as its security implications.

Before adding the masked attributes value to the key generation material, the **EGETKEY** implementation forces the mask bits corresponding to the **INIT** and **DEBUG** attributes (§ 5.2.2) to be set. From a practical standpoint, this means that secrets will never be migrated between enclaves that support debugging and production enclaves.

Without this restriction, it would be unsafe for an enclave author to use the same **RSA** key to issue certificates to both debugging and

production enclaves. Debugging enclaves receive no integrity guarantees from SGX, so it is possible for an attacker to modify the code inside a debugging enclave in a way that causes it to disclose any secrets that it has access to.

## 5.8 SGX Software Attestation

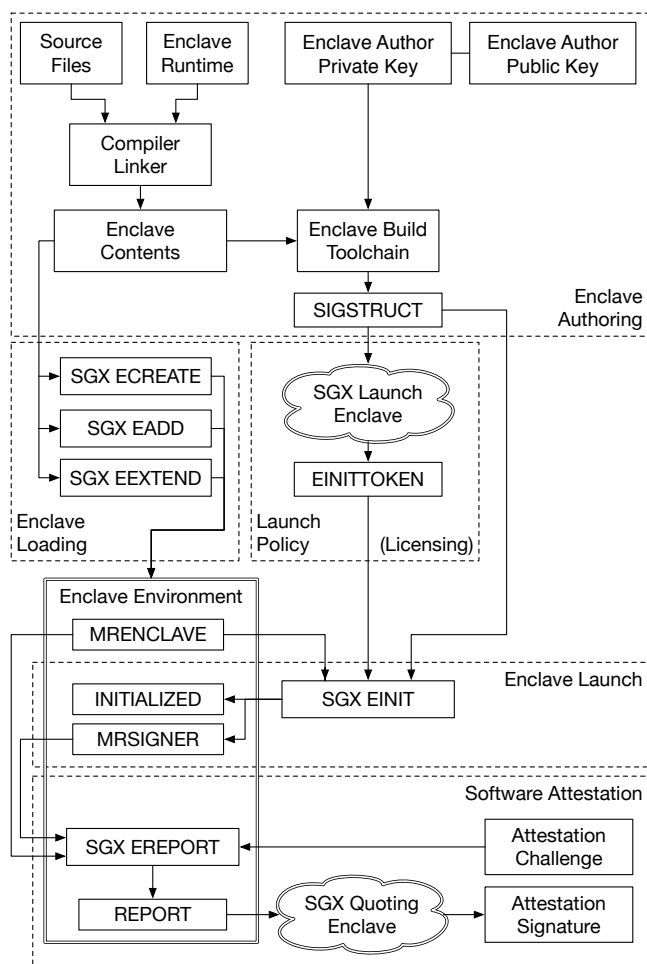
The software attestation scheme implemented by SGX follows the principles outlined in § 3.3, and is illustrated at a high level by Figure 5.20. An SGX-enabled processor computes a measurement of the code and data that is loaded in each enclave, which is similar to the measurement computed by the TPM (§ 4.4). The software inside an enclave can start a process that results in an SGX attestation signature, which includes the enclave's measurement and an enclave message.

The cryptographic primitive used in SGX's attestation signature is too complex to be implemented in hardware, so the signing process is performed by a privileged *Quoting Enclave*, which is issued by Intel, and can access the SGX attestation key. This enclave is discussed in § 5.8.2.

Pushing the signing functionality into the Quoting Enclave creates the need for a secure communication path between an enclave undergoing software attestation and the Quoting Enclave. The SGX design solves this problem with a local attestation mechanism that can be used by an enclave to prove its identity to any other enclave hosted by the same SGX-enabled CPU. This scheme, described in § 5.8.1, is implemented by the `EREPOR`T instruction.

The SGX attestation key used by the Quoting Enclave does not exist at the time SGX-enabled processors leave the factory. The attestation key is provisioned later, using a process that involves a Provisioning Enclave issued by Intel, and two special `EGETKEY` (§ 5.7.5) key types. The publicly available details of this process are summarized in § 5.8.2.

The SGX Launch Enclave and `EINITTOKEN` structure will be discussed in § 5.9.

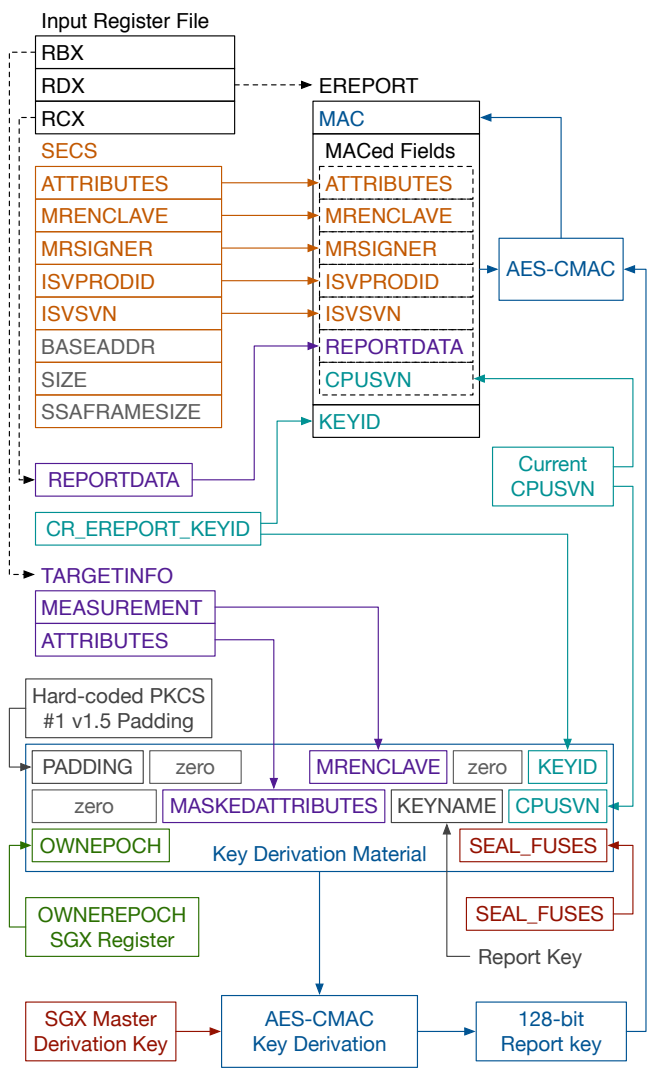


**Figure 5.20:** Setting up an SGX enclave and undergoing the software attestation process involves the SGX instructions EINIT and EREPORT, and two special enclaves authored by Intel, the SGX Launch Enclave and the SGX Quoting Enclave.

### 5.8.1 Local Attestation

An enclave proves its identity to another *target enclave* via the EREPORT instruction shown in Figure 5.21. The SGX instruction produces an attestation *Report* (REPORT) that cryptographically binds a message supplied by the enclave with the enclave’s measurement-based (§ 5.6)

and certificate-based (§ 5.7.2) identities. The cryptographic binding is accomplished by a MAC tag (§ 3.1.3) computed using a symmetric key that is only shared between the target enclave and the SGX implementation.



**Figure 5.21:** EREPORT data flow.

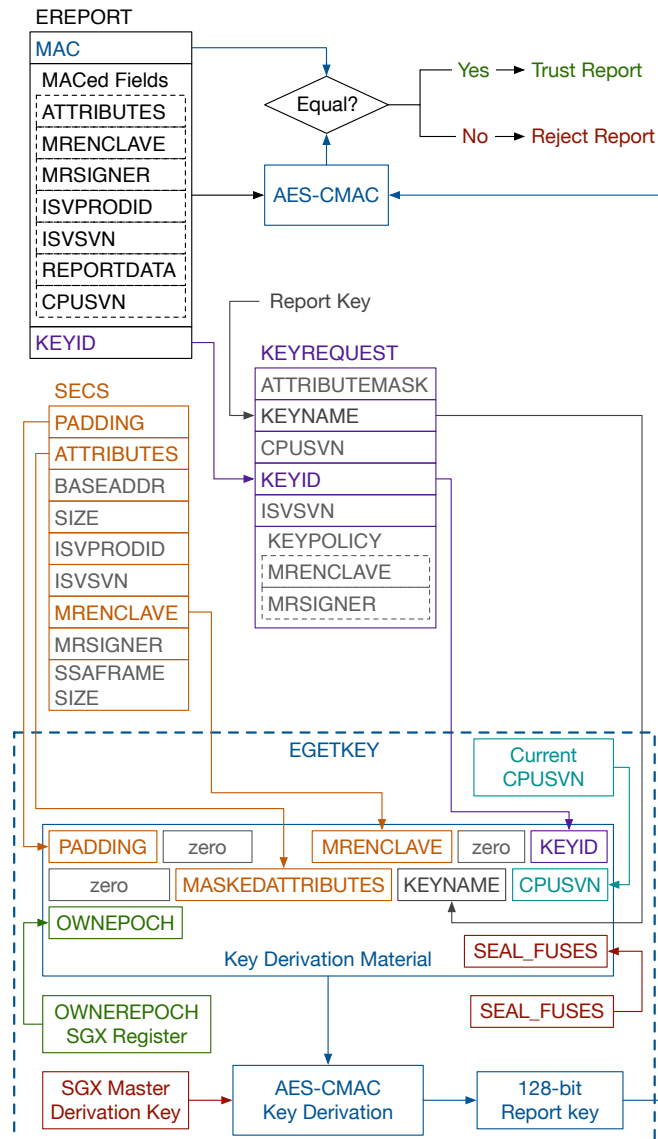
The `EREPOR`T instruction reads the current enclave’s identity information from the enclave’s SECS (§ 5.1.3), and uses it to populate the `REPORT` structure. Specifically, `EREPOR`T copies the SECS fields indicating the enclave’s measurement (`MRENCLAVE`), certificate-based identity (`MRSIGNER`, `ISVPRODID`, `ISVSVN`), and attributes (`ATTRIBUTES`). The attestation report also includes the SVN of the SGX implementation (`CPUSVN`) and a 64-byte (512-bit) message supplied by the enclave.

The target enclave that receives the attestation report can convince itself of the report’s authenticity as shown in Figure 5.22. The report’s authenticity proof is its MAC tag. The key required to verify the MAC can only be obtained by the target enclave, by asking `EGETKEY` (§ 5.7.5) to derive a Report key. The SDM states that the MAC tag is computed using a block cipher-based MAC (`CMAC`, [Dworkin, 2005]), but stops short of specifying the underlying cipher. One of the SGX papers [Anati et al., 2013] states that the `CMAC` is based on 128-bit AES.

The Report key returned by `EGETKEY` is derived from a secret embedded in the processor (§ 5.7.5), and the key material includes the target enclave’s measurement. The target enclave can be assured that the MAC tag in the report was produced by the SGX implementation, for the following reasons. The cryptographic properties of the underlying key derivation and MAC algorithms ensure that only the SGX implementation can produce the MAC tag, as it is the only entity that can access the processor’s secret, and it would be impossible for an attacker to derive the Report key without knowing the processor’s secret. The SGX design guarantees that the key produced by `EGETKEY` depends on the calling enclave’s measurement, so only the target enclave can obtain the key used to produce the MAC tag in the report.

`EREPOR`T uses the same key derivation process as `EGETKEY` does when invoked with `KEYNAME` set to the value associated with Report keys. For this reason, `EREPOR`T requires the virtual address of a *Report Target Info* (`TARGETINFO`) structure that contains the measurement-based identity and attributes of the target enclave.

When deriving a Report key, `EGETKEY` behaves slightly differently than it does in the case of seal keys, as shown in Figure 5.22. The



**Figure 5.22:** The authenticity of the REPORT structure created by EREPORT can and should be verified by the report's target enclave. The target's code uses EGETKEY to obtain the key used for the MAC tag embedded in the REPORT structure, and then verifies the tag.



key generation material never includes the fields corresponding to the enclave's certificate-based identity (MRSIGNER, ISVPRODID, ISVSVN), and the KEYPOLICY field in the KEYREQUEST structure is ignored. It follows that the report can only be verified by the target enclave.

Furthermore, the SGX implementation's SVN (CPUSVN) value used for key generation is determined by the current CPUSVN, instead of being read from the Key Request structure. Therefore, SGX implementation upgrades that increase the CPUSVN invalidate all outstanding reports. Given that CPUSVN increases are associated with security fixes, the argument in § 5.7.2 suggests that this restriction may reduce the impact of vulnerabilities in the SGX implementation.

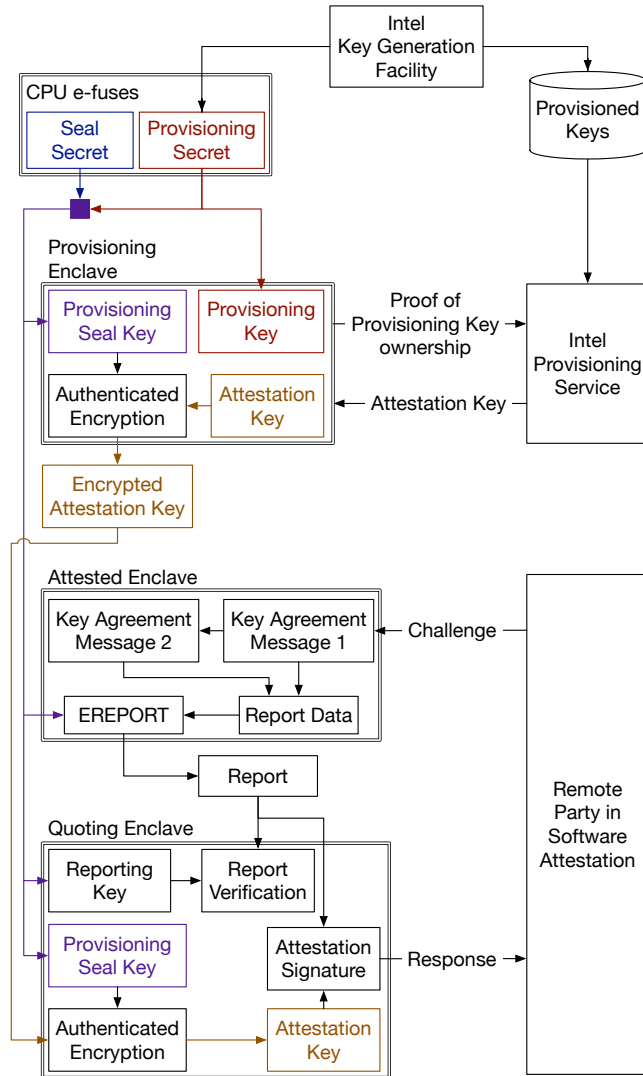
Last, EREPORT sets the KEYID field in the key generation material to the contents of an SGX configuration register (CR\_REPORT\_KEYID) that is initialized with a random value when SGX is initialized. The KEYID value is also saved in the attestation report, but it is not covered by the MAC tag.

### 5.8.2 Remote Attestation

The SDM paints a complete picture of the local attestation mechanism that was described in § 5.8.1. The remote attestation process, which includes the Quoting Enclave and the underlying keys, is covered at a high level in an Intel publication [Johnson et al., 2016]. This section's contents is based on the SDM, on one [Anati et al., 2013] of the SGX papers, and on the ISCA 2015 SGX tutorial [Int, 2015f].

SGX's software attestation scheme, which is illustrated in Figure 5.23, relies on a key generation facility and on a provisioning service, both operated by Intel.

During the manufacturing process, an SGX-enabled processor communicates with Intel's key generation facility, and has two secrets burned into e-fuses, which are a one-time programmable storage medium that can be economically included on a high-performance chip's die. We shall refer to the secrets stored in e-fuses as the *Provisioning Secret* and the *Seal Secret*.



**Figure 5.23:** SGX's software attestation is based on two secrets stored in e-fuses inside the processor's die, and on a key received from Intel's provisioning service.

The Provisioning Secret is the main input to a largely undocumented process that outputs the SGX master derivation key used by **EGETKEY**, which was referenced in Figures 5.19, 5.20, 5.21, and 5.22.

The Seal Secret is not exposed to software by any of the architectural mechanisms documented in the SDM. The secret is only accessed when it is included in the material used by the key derivation process implemented by **EGETKEY** (§ 5.7.5). The pseudocode in the SDM uses the `CR_SEAL_FUSES` register name to refer to the Seal Secret.

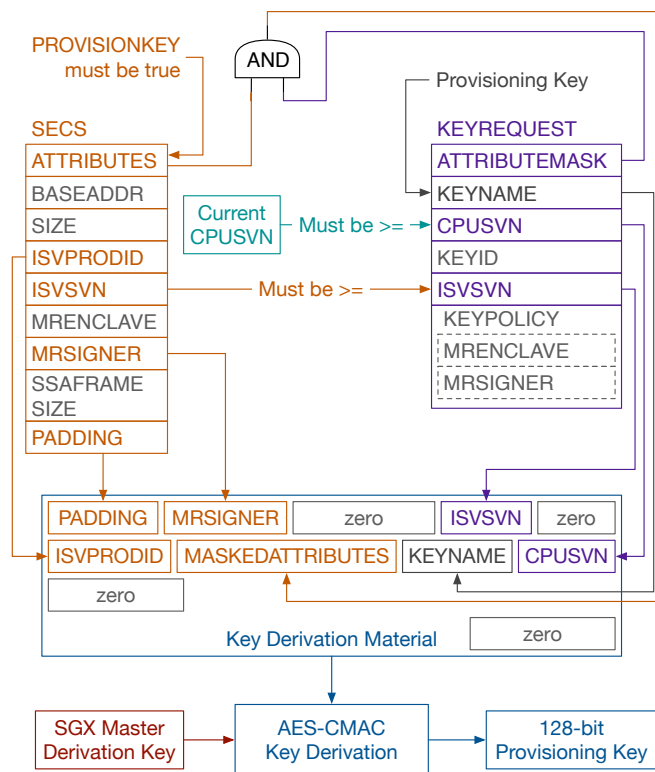
The names “Seal Secret” and “Provisioning Secret” deviate from Intel’s official documents, which confusingly use the “Seal Key” and “Provisioning Key” names to refer to both secrets stored in e-fuses and keys derived by **EGETKEY**.

The SDM briefly describes the keys produced by **EGETKEY**, but no official documentation explicitly describes the secrets in e-fuses. The description below is the only interpretation of all public information sources that is consistent with all statements in the SDM regarding key derivation.

The Provisioning Secret is generated at the key generation facility, where it is burned into the processor’s e-fuses and stored in the database used by Intel’s provisioning service. The Seal Secret is generated inside the processor chip, and therefore is not known to Intel. This approach has the benefit that an attacker who compromises Intel’s facilities cannot derive most keys produced by **EGETKEY**, even if the attacker also compromises a victim’s firmware and obtains the `OWNEREPOCH` (§ 5.7.5) value. These keys include the Seal keys (§ 5.7.5) and Report keys (§ 5.8.1) introduced in previous sections.

The only documented exception to the reasoning above is the *Provisioning key*, which is effectively a shared secret between the SGX-enabled processor and Intel’s provisioning service. Intel has to be able to derive this key, so the derivation material does not include the Seal Secret or the `OWNEREPOCH` value, as shown in Figure 5.24.

**EGETKEY** derives the Provisioning key using the current enclave’s certificate-based identity (`MRSIGNER`, `ISVPRODID`, `ISVSVN`) and the SGX implementation’s SVN (`CPUSVN`). This approach has a few desirable security properties. First, Intel’s provisioning service can be



**Figure 5.24:** When EGETKEY is asked to derive a Provisioning key, it does not use the Seal Secret or OWNEREPOCH. The Provisioning key does, however, depend on MRSIGNER and on the SVN of the SGX implementation.

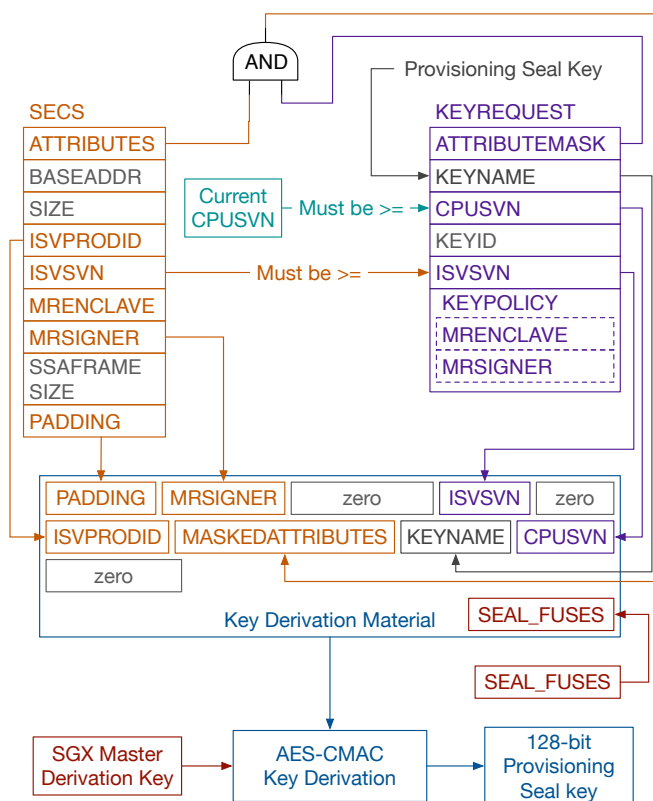
assured that it is authenticating a Provisioning Enclave signed by Intel. Second, the provisioning service can use the CPUSVN value to reject SGX implementations with known security vulnerabilities. Third, this design admits multiple mutually distrusting provisioning services.

EGETKEY only derives Provisioning keys for enclaves whose PROVISIONKEY attribute is set to true. § 5.9.3 argues that this mechanism is sufficient to protect the computer owner from a malicious software provider that attempts to use Provisioning keys to track a processor across OWNEREPOCH changes.

After the Provisioning Enclave obtains a Provisioning key, it uses the key to authenticate itself to Intel's provisioning service. Once the

provisioning service is convinced that it is communicating to a trusted Provisioning enclave in the secure environment provided by a SGX-enabled processor, the service generates an *Attestation Key* and sends it to the Provisioning Enclave. The enclave then encrypts the Attestation Key using a *Provisioning Seal key*, and hands off the encrypted key to the system software for storage.

Provisioning Seal keys, are the last publicly documented type of special keys derived by EGETKEY, using the process illustrated in Figure 5.25. As their name suggests, Provisioning Seal keys are conceptually similar to the Seal Keys (§ 5.7.5) used to migrate secrets between enclaves.



**Figure 5.25:** The derivation material used to produce Provisioning Seal keys does not include the OWNEREPOCH value, so the keys survive computer ownership changes.

The defining feature of Provisioning Seal keys is that they are not based on the `OWNEREPOCH` value, so they survive computer ownership changes. Since Provisioning Seal keys can be used to track a processor, their use is gated on the `PROVISIONKEY` attribute, which has the same semantics as for Provisioning keys.

Like Provisioning keys, Seal keys are based on the current enclave's certificate-based identity (`MRSIGNER`, `ISVPROD`, `ISVSVN`), so the Attestation Key encrypted by Intel's Provisioning Enclave can only be decrypted by another enclave signed with the same Intel RSA key. However, unlike Provisioning keys, the Provisioning Seal keys are based on the Seal Secret in the processor's e-fuses, so they cannot be derived by Intel.

When considered independently from the rest of the SGX design, Provisioning Seal keys have desirable security properties. The main benefit of these keys is that when a computer with an SGX-enabled processor exchanges owners, it does not need to undergo the provisioning process again, so Intel does not need to be aware of the ownership change. The confidentiality issue that stems from not using `OWNEREPOCH` was already introduced by Provisioning keys, and is mitigated using the access control scheme based on the `PROVISIONKEY` attribute that will be discussed in § 5.9.3.

Similarly to the Seal key derivation process, both the Provisioning and Provisioning Seal keys depend on the bitwise AND of the `ATTRIBUTES` (§ 5.2.2) field in the enclave's `SECS` and the `ATTRIBUTESMASK` field in the `KEYREQUEST` structure. While most attributes can be masked away, the `DEBUG` and `INIT` attributes are always used for key derivation.

This dependency makes it safe for Intel to use its production RSA key to issue certificates for Provisioning or Quoting Enclaves with debugging features enabled. Without the forced dependency on the `DEBUG` attribute, using the production Intel signing key on a single debug Provisioning or Quoting Enclave could invalidate SGX's security guarantees on all CPU devices whose attestation-related enclaves are signed by the same key. Concretely, if the issued `SIGSTRUCT` would be leaked, any attacker could build a debugging Provisioning or Quot-

ing enclave, use the SGX debugging features to modify the code inside it, and extract the 128-bit Provisioning key used to authenticated the CPU to Intel’s provisioning service.

After the provisioning steps above have been completed, the Quoting Enclave can be invoked to perform SGX’s software attestation. This enclave receives local attestation reports (§ 5.8.1) and verifies them using the Report keys generated by **EGETKEY**. The Quoting Enclave then obtains the Provisioning Seal Key from **EGETKEY** and uses it to decrypt the Attestation Key, which is received from system software. Last, the enclave replaces the MAC in the local attestation report with an *Attestation Signature* produced with the Attestation Key.

The SGX patents state that the name “Quoting Enclave” was chosen as a reference to the TPM (§ 4.4)’s quoting feature, which is used to perform software attestation on a TPM-based system.

The Attestation Key uses Intel’s *Enhanced Privacy ID* (EPID) cryptosystem [Brickell and Li, 2009], which is a group signature scheme that is intended to preserve the anonymity of the signers. Intel’s key provisioning service is the issuer in the EPID scheme, so it publishes the Group Public Key, while securely storing the Master Issuing Key. After a Provisioning Enclave authenticates itself to the provisioning service, it generates an EPID Member Private Key, which serves as the Attestation Key, and executes the EPID Join protocol to join the group. Later, the Quoting Enclave uses the EPID Member Private Key to produce Attestation Signatures.

The Provisioning Secret stored in the e-fuses of each SGX-enabled processor can be used by Intel to trace individual physical processor packages when a Provisioning Enclave authenticates itself to the provisioning service. However, if the EPID Join protocol is blinded, Intel’s provisioning service cannot trace an Attestation Signature to a specific Attestation Key, so Intel cannot trace Attestation Signatures to individual CPUs.

Of course, the security properties of the description above hinge on the correctness of the proofs behind the EPID scheme. Analyzing the correctness of such cryptographic schemes is beyond the scope

of this work, so we defer the analysis of EPID to the crypto research community.

## 5.9 SGX Enclave Launch Control

The SGX design includes a launch control process, which introduces an unnecessary approval step that is required before running most enclaves on a computer. The approval decision is made by the *Launch Enclave* (LE), which is an enclave issued by Intel that gets to approve every other enclave before it is initialized by *EINIT* (§ 5.3.3). The officially documented information about this approval process is discussed in § 5.9.1.

The SGX patents [McKeen et al., 2009, Johnson et al., 2010] disclose in no uncertain terms that the Launch Enclave was introduced to ensure that each enclave's author has a business relationship with Intel, and implements a software licensing system. § 5.9.2 briefly discusses the implications, should this turn out to be true.

The remainder of the section argues that the Launch Enclave should be removed from the SGX design. § 5.9.3 explains that the LE is not required to enforce the computer owner's launch control policy, and concludes that the LE is only meaningful if it enforces a policy that is detrimental to the computer owner. § 5.9.4 debunks the myth that an enclave can host malware, which is likely to be used to justify the LE. § 5.9.5 argues that Anti-Virus (AV) software is not fundamentally incompatible with enclaves, further disproving the theory that Intel needs to actively police the software that runs inside enclaves.

### 5.9.1 Enclave Attributes Access Control

The SGX design requires that all enclaves be vetted by a Launch Enclave (LE), which is only briefly mentioned in Intel's official documentation. Neither its behavior nor its interface with the system software is specified. We speculate that Intel has not been forthcoming about the LE because of its role in enforcing software licensing, which will be discussed in § 5.9.2. This section abstracts away the licensing aspect and assumes that the LE enforces a black-box Launch Control Policy.



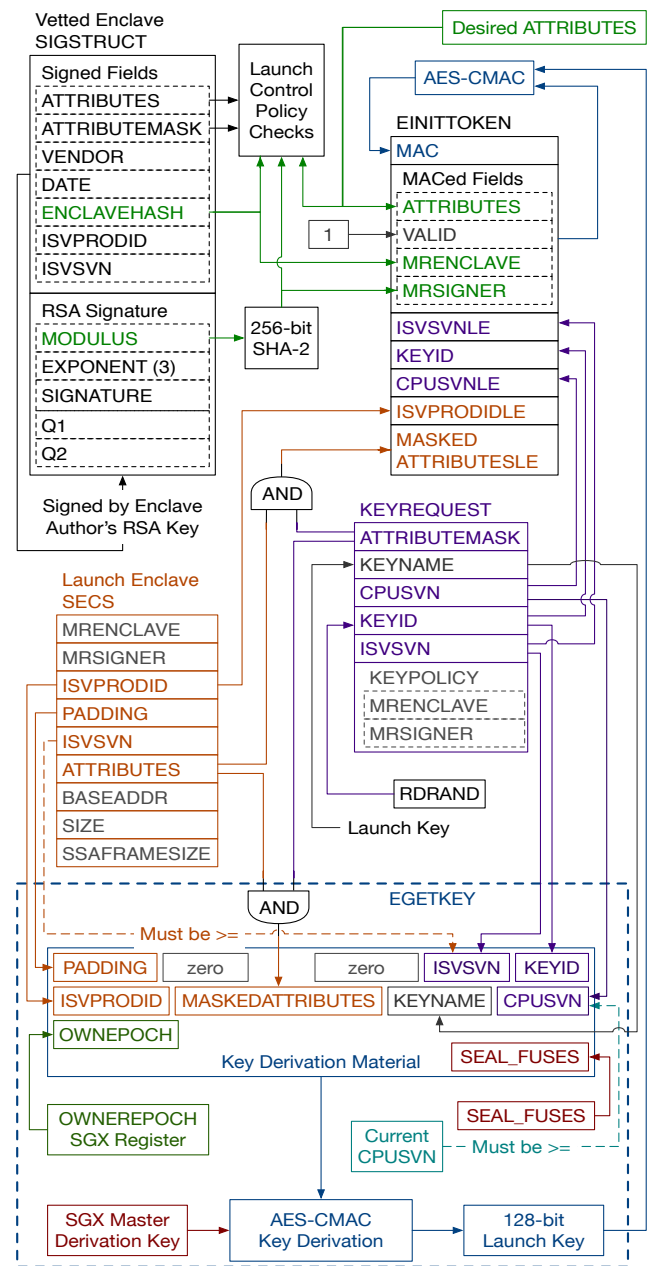
The LE approves an enclave by issuing an *EINIT Token* (EINITTOKEN), using the process illustrated in Figure 5.26. The EINITTOKEN structure contains the approved enclave’s measurement-based (§ 5.6) and certificate-based (§ 5.7.2) identities, just like a local attestation REPORT (§ 5.8.1). This token is inspected by EINIT (§ 5.3.3), which refuses to initialize enclaves with incorrect tokens.

While an EINIT token is handled by untrusted system software, its integrity is protected by a MAC tag (§ 3.1.3) that is computed using a *Launch Key* obtained from EGETKEY. The EINIT implementation follows the same key derivation process as EGETKEY to convince itself that the EINITTOKEN provided to it was indeed generated by an LE that had access to the Launch Key.

The SDM does not document the MAC algorithm used to confer integrity guarantees to the EINITTOKEN structure. However, the EINIT pseudocode verifies the token’s MAC tag using the same function that the *EREPORT* pseudocode uses to create the REPORT structure’s MAC tag. It follows that the reasoning in § 5.8.1 can be reused to conclude that EINITTOKEN structures are MACed using AES-CMAC with 128-bit keys.

The EGETKEY instruction only derives the Launch Key for enclaves that have the LAUNCHKEY attribute set to true. The Launch Key is derived using the same process as the Seal Key (§ 5.7.5). The derivation material includes the current enclave’s versioning information (ISVPRODID and ISVSVN) but it does not include the main fields that convey an enclave’s identity, which are MRSIGNER and MRENCLAVE. The rest of the derivation material follows the same rules as the material used for Seal Keys.

The EINITTTOKEN structure contains the identities of the approved enclave (MRENCLAVE and MRSIGNER) and the approved enclave attributes (ATTRIBUTES). The token also includes the information used for the Launch Key derivation, which includes the LE’s Product ID (ISVPRODIDLE), SVN (ISVSVNLE), and the bitwise AND between the LE’s ATTRIBUTES and the ATTRIBUTEMASK used in the KEYREQUEST (MASKEDATTRIBUTESLE).



**Figure 5.26:** The SGX Launch Enclave computes the EINITTOKEN.

The EINITTOKEN information used to derive the Launch Key can also be used by EINIT for damage control, e.g. to reject tokens issued by Launch Enclaves with known security vulnerabilities. The reference pseudocode supplied in the SDM states that EINIT checks the DEBUG bit in the MASKEDATTRIBUTESLE field, and will not initialize a production enclave using a token issued by a debugging LE. It is worth noting that MASKEDATTRIBUTESLE is guaranteed to include the LE's DEBUG attribute, because EGETKEY forces the DEBUG attribute's bit in the attributes mask to 1 (§ 5.7.5).

The check described above make it safe for Intel to supply SGX enclave developers with a debugging LE that has its DEBUG attribute set, and performs minimal or no security checks before issuing an EINITTOKEN. The DEBUG attribute disables SGX's integrity protection, so the only purpose of the security checks performed in the debug LE would be to help enclave development by mimicking its production counterpart. The debugging LE can only be used to launch any enclave with the DEBUG attribute set, so it does not undermining Intel's ability to enforce a Launch Control Policy on production enclaves.

The enclave attributes access control system described above relies on the LE to reject initialization requests that set privileged attributes such as PROVISIONKEY on unauthorized enclaves. However, the LE cannot vet itself, as there will be no LE available when the LE itself needs to be initialized. Therefore, the Launch Key access restrictions are implemented in hardware.

EINIT accepts an EINITTOKEN whose VALID bit is set to zero, if the enclave's MRSIGNER (§ 5.7.1) equals a hard-coded value that corresponds to an Intel public key. For all other enclave authors, an invalid EINIT token causes EINIT to reject the enclave and produce an error code.

This exemption to the token verification policy provides a way to bootstrap the enclave attributes access control system, namely using a zeroed out EINITTOKEN to initialize the Launch Enclave. At the same time, the cryptographic primitives behind the MRSIGNER check guarantee that only Intel-provided enclaves will be able to bypass the attribute checks. This does not change SGX's security prop-

erties because Intel is already a trusted party, as it is responsible for generating the Provisioning Keys and Attestation Keys used by software attestation (§ 5.8.2).

Curiously, the EINIT pseudocode in the SDM states that the instruction enforces an additional restriction, which is that all enclaves with the LAUNCHKEY attribute must have their certificates issued by the same Intel public key that is used to bypass the EINITTOKEN checks. This restriction appears to be redundant, as the same restriction could be enforced in the Launch Enclave.

### 5.9.2 Licensing

The SGX patents [McKeen et al., 2009, Johnson et al., 2010] disclose that EINIT Tokens and the Launch Enclave (§ 5.9.1) were introduced to verify that the SIGSTRUCT certificates associated with production enclaves are issued by enclave authors who have a business relationship with Intel. In other words, the Launch Enclave is intended to be **an enclave licensing mechanism that allows Intel to force itself as an intermediary in the distribution of all enclave software**.

The SGX patents are likely to represent an early version of the SGX design, due to the lengthy timelines associated with patent application approval. In light of this consideration, we cannot make any claims about Intel's current plans. However, given that we know for sure that Intel considered enclave licensing at some point, we briefly discuss the implications of implementing such a licensing plan.

Intel has a near-monopoly on desktop and server-class processors, and being able to decide which software vendors are allowed to use SGX can effectively put Intel in a position to decide winners and losers in many software markets.

Assuming SGX reaches widespread adoption, this issue is the software security equivalent to the Net Neutrality debates that have pitted the software industry against telecommunication giants. Given that virtually all competent software development companies have argued that losing Net Neutrality will stifle innovation, it is fairly safe to assume that Intel's ability to regulate access to SGX will also stifle innovation.

Furthermore, from a historical perspective, the enclave licensing scheme described in the SGX patents is very similar to Verified Boot, which was briefly discussed in § 4.4. Verified Boot has mostly received negative reactions from software developers, so it is likely that an enclave licensing scheme would meet the same fate, should the developer community become aware of it.

### 5.9.3 System Software Can Enforce a Launch Policy

§ 5.3 explains that the SGX instructions used to load and initialize enclaves (`ECREATE`, `EADD`, `EINIT`) can only be issued by privileged system software, because they manage the EPC, which is a system resource.

A consequence on the restriction that only privileged software can issue `ECREATE` and `EADD` instructions is that the system software is able to track all public information that is loaded into each enclave. The privilege requirements of `EINIT` mean that the system software can also examine each enclave's `SIGSTRUCT`. It follows that the system software has access to a superset of the information that the Launch Enclave may use.

Furthermore, `EINIT`'s privileged instruction status means that the system software can perform its own policy checks before allowing application software to initialize an enclave. So, the system software can enforce a Launch Control Policy set by the computer's owner. For example, an IaaS cloud service provider may use its hypervisor to implement a Launch Control Policy that limits what enclaves its customers are allowed to execute.

Given that the system software has access to a superset of the information that the Launch Enclave may use, it is easy to see that the set of policies that can be enforced by system software is a superset of the policies that can be supported by an LE. Therefore, the only rational explanation for the existence of the LE is that it was designed to implement a Launch Control Policy that is not beneficial to the computer owner.

As an illustration of this argument, we consider the case of restricting access to `EGETKEY`'s Provisioning keys (§ 5.8.2). The derivation material for Provisioning keys does not include `OWNEREPOCH`, so malicious enclaves can potentially use these keys to track a CPU

chip package as it exchanges owners. For this reason, the SGX design includes a simple access control mechanism that can be used by system software to limiting enclave access to Provisioning keys. `EGETKEY` refuses to derive Provisioning keys for enclaves whose `PROVISIONKEY` attribute is not set to true.

It follows that a reasonable Launch Control Policy would only allow the `PROVISIONKEY` attribute to be set for the enclaves that implement software attestation, such as Intel's Provisioning Enclave and Quoting Enclave. This policy can easily be implemented by system software, given its exclusive access to the `EINIT` instruction.

The only concern with the approach outlined above is that a malicious system software may abuse the `PROVISIONKEY` attribute to generate a unique identifier for the hardware that it runs on, similar to the much maligned Intel Processor Serial Number [Int, 1999]. We dismiss this concern by pointing out that system software has access to many unique identifiers, such as the Media Access Control (MAC) address of the Ethernet adapter integrated into the motherboard's chipset (§ 2.9.1).

#### **5.9.4 Enclaves Cannot Damage the Host Computer**

SGX enclaves execute at the lowest privilege level (user mode / ring 3), so they are subject to the same security checks as their host application. For example, modern operating systems set up the I/O maps (§ 2.7) to prevent application software from directly accessing the I/O address space (§ 2.4), and use the supervisor (S) page table attribute (§ 2.5.3) to deny application software direct access to memory-mapped devices (§ 2.4) and to the DRAM that stores the system software. Enclave software is subject to I/O privilege checks and address translation checks, so a malicious enclave cannot directly interact with the computer's devices, and cannot tamper the system software.

It follows that software running in an enclave has the same means to compromise the system software as its host application, which come down to exploiting a security vulnerability. The same solutions used to mitigate vulnerabilities exploited by application software (e.g., `seccomp/bpf` [Kim and Zeldovich, 2013]) apply to enclaves.

The only remaining concern is that an enclave can perform a denial of service (DoS) attack against the system software. The rest of this section addresses the concern.

The SGX design provides system software the tools it needs to protect itself from enclaves that engage in CPU hogging and DRAM hogging. As enclaves cannot perform I/O directly, these are the only two classes of DoS attacks available to them.

An enclave that attempts to hog an LP assigned to it can be preempted by the system software via an Inter-Processor Interrupt (IPI, § 2.12) issued from another processor. This method is available as long as the system software reserves at least one LP for non-enclave computation.

Furthermore, most OS kernels use tick schedulers, which use a real-time clock (RTC) configured to issue periodical interrupts (ticks) to all cores. The RTC interrupt handler invokes the kernel's scheduler, which chooses the thread that will get to use the logical processor until the next RTC interrupt is received. Therefore, kernels that use tick schedulers always have the opportunity to de-schedule enclave threads, and don't need to rely on the ability to send IPIs.

In SGX, the system software can always evict an enclave's EPC pages to non-EPC memory, and then to disk. The system software can also outright deallocate an enclave's EPC pages, though this will probably cause the enclave code to encounter page faults that cannot be resolved. The only catch is that the EPC pages that hold metadata for running enclave threads cannot be evicted or removed. However, this can easily be resolved, as the system software can always preempt enclave threads, using one of the methods described above.

### 5.9.5 Interaction with Anti-Virus Software

Today's anti-virus (AV) systems are glorified pattern matchers. AV software simply scans all executable files on the system and the memory of running processes, looking for bit patterns that are thought to only occur in malicious software. These patterns are somewhat pompously called "virus signatures".

SGX (and TXT, to some extent) provides a method for executing code in an isolated container that we refer to as an enclave. Enclaves are isolated from all other software on the computer, including any AV software that may be installed.

The isolation afforded by SGX opens up the possibility for bad actors to structure their attacks as a generic loader that would end up executing a malicious payload without tripping the AV's pattern matcher. More specifically, the attack would create an enclave and initialize it with a generic loader that looks innocent to an AV. The loader inside the enclave would obtain an encrypted malicious payload, and would undergo software attestation with an Internet server to obtain the payload's encryption key. The loader would then decrypt the malicious payload and execute it inside the enclave.

In the scheme suggested here, the malicious payload only exists in a decrypted form inside an enclave's memory, which cannot be accessed by the AV. Therefore, the AV's pattern matcher will not trip.

This issue does not have a solution that maintains the status-quo for the AV vendors. The attack described above would be called a protection scheme if the payload would be a proprietary image processing algorithm, or a DRM scheme.

On a brighter note, enclaves do not bring the complete extinction of AV, they merely require a change in approach. Enclave code always executes at the lowest privilege mode (ring 3 / user mode), so it cannot perform any I/O without invoking the services of system software. For all intents and purposes, this effectively means that enclave software cannot perform any malicious action without the complicity of system software. Therefore, enclaves can be policed effectively by intelligent AV software that records and filters the I/O performed by software, and detects malicious software according to the actions that it performs, rather than according to bit patterns in its code.

Furthermore, SGX's enclave loading model allows the possibility of performing static analysis on the enclave's software. For simplicity, assume the existence of a standardized static analysis framework. The initial enclave contents is not encrypted, so the system software can easily perform static analysis on it. Dynamically loaded code or Just-In-Time



code generation (JIT) can be handled by requiring that all enclaves that use these techniques embed the static analysis framework and use it to analyze any dynamically loaded code before it is executed. The system software can use static verification to ensure that enclaves follow these rules, and refuse to initialize any enclaves that fail verification.

In conclusion, enclaves in and of themselves don't introduce new attack vectors for malware. However, the enclave isolation mechanism is fundamentally incompatible with the approach employed by today's AV solutions. Fortunately, it is possible (though non-trivial) to develop more intelligent AV software for enclave software.

# 6

---

## Conclusion

---

This manuscript is the first of a two-part review of secure processor systems that aims to enable remote computation with guarantees of privacy and integrity. § 2 introduced computer architecture concepts relevant to the work’s discussion of trusted remote computation, and enclaves in particular. An understanding of the intended and unintended properties of virtual memory, cache hierarchies, fine-grained multithreading, and data structures managed by Operating System is instrumental to a rigorous discussion of the security properties of a trusted system. § 3 discussed practical cryptographic primitives and protocols, as well as attack vectors exposed by modern computer systems. It provided concrete context against the threat models employed by secure processor systems can be evaluated. § 4 is a brief survey of a large body of secure processor systems, including commentary on their threat models, design decisions, and success against real-world adversaries. Finally, § 5 presented a practical approach to and programming model for a secure enclave with a small trusted computing base.

Part II of this work presents a deep dive into the design decisions and resulting quirks of SGX and an analysis of the system’s security properties and threat model. Given this discussion, the work

will present MIT's Sanctum, an enclave-capable secure system with a stronger security argument under a software threat model than SGX.

While this work does not seek to prescribe any specific solution to the security needs of a given application, we invite the reader to carefully examine the software and hardware included in the trusted computing base of the services they rely on. Security implies some overhead, and the tradeoff between cost, performance, design effort, and security must be carefully considered in any application. A claim of security of a given system is meaningless without a corresponding threat model and rigorous analysis of the system's trusted computing base. With a principled, transparent, and well-scrutinized approach to system design, practical guarantees of privacy and integrity for remote computation are well within reach.

## 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

---

- FIPS 140-2 Consolidated Validation Certificate No. 0003*. 2011.
- IBM 4765 Cryptographic Coprocessor Security Module - Security Policy*. Dec 2012.
- 7-Zip LZMA benchmark: Intel Haswell. <http://www.7-cpu.com/cpu/Haswell.html>, 2014. [Online; accessed 10-February-2015].
- Linux kernel: CVE security vulnerabilities, versions and detailed reports. [http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor\\_id=33](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](http://www.cvedetails.com/product/23463/XEN-XEN.html?vendor_id=6276), 2014b. [Online; accessed 27-April-2015].
- IPC2 hardware specification. <http://fit-pc.com/download/intense-pc2/documents/ipc2-hw-specification.pdf>, Sep 2014. [Online; accessed 2-Dec-2015].
- Gradually sunseting SHA-1. <http://googleonlinesecurity.blogspot.com/2014/09/gradually-sunseting-sha-1.html>, 2014. [Online; accessed 4-May-2015].
- NIST'S policy on hash functions. <http://csrc.nist.gov/groups/ST/hash/policy.html>, 2014. [Online; accessed 4-May-2015].
- BIOS freedom status. <https://puri.sm/posts/bios-freedom-status/>, Nov 2014. [Online; accessed 2-Dec-2015].
- Xen project software overview. [http://wiki.xen.org/wiki/Xen\\_Project\\_Software\\_Overview](http://wiki.xen.org/wiki/Xen_Project_Software_Overview), 2015. [Online; accessed 27-April-2015].

- SHA-1 deprecation countdown. <https://blogs.windows.com/msedgedev/2016/11/18/countdown-to-sha-1-deprecation/#MPDwCxdpw3IqPPBR>. 97, 2016. [Online; accessed 18-June-2017].
- Seth Abraham. Time to revisit REP;MOVS - comment. <https://software.intel.com/en-us/forums/topic/275765>, Aug 2006. [Online; accessed 23-January-2015].
- Tiago Alves and Don Felton. TrustZone: Integrated hardware and software security. *Information Quarterly*, 3(4):18–24, 2004.
- 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.
- Ross Anderson. *Security engineering: A guide to building dependable distributed systems*. Wiley, 2001.
- 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].
- AMBA® AXI Protocol. ARM Limited, Mar 2004. Reference no. IHI 0022B, IHI 0024B, AR500-DA-10004.
- ARM Security Technology Building a Secure System using TrustZone® Technology. ARM Limited, Apr 2009. Reference no. PRD29-GENC-009492C.
- Sebastian Banescu. Cache timing attacks. 2011. [Online; accessed 26-January-2014].
- Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management – part 1: General (revision 3). *Federal Information Processing Standards (FIPS) Special Publications (SP)*, 800-57, Jul 2012.
- Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Secure hash standard (SHS). *Federal Information Processing Standards (FIPS) Publications (PUBS)*, 180-4, Aug 2015.
- Friedrich Beck. *Integrated Circuit Failure Analysis: a Guide to Preparation Techniques*. John Wiley & Sons, 1998.
- Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Advances in Cryptology – CRYPTO’98*, pages 1–12. Springer, 1998.

- D. D. Boggs and S. D. Rodgers. Microprocessor with novel instruction for signaling event occurrence and for providing event handling information in response thereto, 1997. US Patent 5,625,788.
- Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 201–215. Springer, 2006.
- 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.
- John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. BIOS chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, pages 25–36. ACM, 2013.
- David Champagne and Ruby B. Lee. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.
- Daming D. Chen and Gail-Joon Ahn. Security analysis of x86 processor microcode. 2014. [Online; accessed 7-January-2015].
- 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.
- Lily Chen. Recommendation for key derivation using pseudorandom functions. *Federal Information Processing Standards (FIPS) Special Publications (SP)*, 800-108, Oct 2009.
- Coreboot. Developer manual, Sep 2014. [Online; accessed 4-March-2015].
- M. P. Cornaby and B. Chaffin. Microinstruction pointer stack including speculative pointers for out-of-order execution, 2007. US Patent 7,231,511.
- Intel Corporation. *Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Reference Manual*, Sep 2014. Reference no. 331051-001.
- Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. *Cryptology ePrint Archive*, Report 2015/564, 2015.

- Victor Costan, Ilia Lebedev, and Srinivas Devadas. Secure processors part II: Intel SGX security analysis and MIT sanctum architecture. In *FnTEDA*, 2017.
- J. Daemen and V. Rijmen. AES proposal: Rijndael, AES algorithm submission, Sep 1999.
- S. M. Datta and M. J. Kumar. Technique for providing secure firmware, 2013. US Patent 8,429,418.
- S. M. Datta, V. J. Zimmer, and M. A. Rothman. System and method for trusted early boot flow, 2010. US Patent 7,752,428.
- Pete Dice. Booting an Intel architecture system, part i: Early initialization. *Dr. Dobbs's*, Dec 2011. [Online; accessed 2-Dec-2015].
- Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.
- Loïc Dufлот, Daniel Etiemble, and Olivier Grumelard. Using CPU system management mode to circumvent operating system security functions. *CanSecWest/core06*, 2006.
- Morris Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. *Federal Information Processing Standards (FIPS) Special Publications (SP)*, 800-38A, Dec 2001.
- Morris Dworkin. Recommendation for block cipher modes of operation: The CMAC mode for authentication. *Federal Information Processing Standards (FIPS) Special Publications (SP)*, 800-38B, May 2005.
- Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. *Federal Information Processing Standards (FIPS) Special Publications (SP)*, 800-38D, Nov 2007.
- D. Eastlake and P. Jones. RFC 3174: US Secure Hash Algorithm 1 (SHA1). *Internet RFCs*, 2001.
- Shawn Embleton, Sherri Sparks, and Cliff C. Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 2010.
- Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. John Wiley & Sons, 2011.
- 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.



- Agner Fog. Instruction tables - lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Dec 2014. [Online; accessed 23-January-2015].
- Andrew Furtak, Yuriy Bulygin, Oleksandr Bazhaniuk, John Loucaides, Alexander Matrosov, and Mikhail Gorobets. BIOS and secure boot attacks uncovered. *The 10th ekoparty Security Conference*, 2014. [Online; accessed 22-October-2015].
- William Futral and James Greene. *Intel® Trusted Execution Technology for Server Platforms*. Apress Open, 2013.
- 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.
- Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 295–306. IEEE, 2003.
- Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013.
- Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs. Cryptology ePrint Archive, Report 2014/626, 2014.
- Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. Cryptology ePrint Archive, Report 2015/170, 2015.
- Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
- R. T. George, J. W. Brandt, K. S. Venkatraman, and S. P. Kim. Dynamically partitioning pipeline resources, 2009. US Patent 7,552,255.
- A. Glew, G. Hinton, and H. Akkary. Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions, 1997. US Patent 5,680,565.
- A. F. Glew, H. Akkary, R. P. Colwell, G. J. Hinton, D. B. Papworth, and M. A. Fetterman. Method and apparatus for implementing a non-blocking translation lookaside buffer, 1996. US Patent 5,564,111.
- 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.

- J. R. Goodman and H. H. J. Hum. MESIF: A two-hop cache coherency protocol for point-to-point interconnects. 2009.
- Joe Grand. Advanced hardware hacking techniques, Jul 2004.
- David Grawrock. *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.
- Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. *CoRR*, abs/1507.06955, 2015. URL <http://arxiv.org/abs/1507.06955>.
- Shay Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016.
- Ben Hawkes. Security analysis of x86 processor microcode. 2012. [Online; accessed 7-January-2015].
- John L. Hennessy and David A. Patterson. *Computer Architecture - a Quantitative Approach (5 ed.)*. Morgan Kaufmann, 2012. ISBN 978-0-12-383872-8.
- Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES smart card implementation resistant to power analysis attacks. In *Applied cryptography and Network security*, pages 239–252. Springer, 2006.
- G. Hildesheim, I. Anati, H. Shafi, S. Raikin, G. Gerzon, U. R. Savagaonkar, C. V. Rozas, F. X. McKeen, M. A. Goldsmith, and D. Prashant. Apparatus and method for page walk extension for enhanced security checks, 2014. US Patent App. 13/730,563.
- 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.
- Gael Hofemeier. Intel manageability firmware recovery agent. Mar 2013. [Online; accessed 2-Dec-2015].
- George Hotz. PS3 glitch hack. 2010. [Online; accessed 7-January-2015].
- Andrew Huang. *Hacking the Xbox: an Introduction to Reverse Engineering*. No Starch Press, 2003.
- C. J. Hughes, Y. K. Chen, M. Bomb, J. W. Brandt, M. J. Buxton, M. J. Charney, S. Chennupaty, J. Corbal, M. G. Dixon, M. B. Girkar, Jonathan C. Hall, Hideki (Saito) Ido, Peter Lachner, Gilbert Neiger, Chris J. Newburn, Rajesh S. Parthasarathy, Bret L. Toll, Robert Valentine, and Jeffrey G. Wiedemeier. Gathering and scattering multiple data elements, 2013. US Patent 8,447,962.

- IEEE Standard for Ethernet*. IEEE Computer Society, Dec 2012. IEEE Std. 802.3-2012.
- 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.
- Intel® Processor Serial Number*. Intel Corporation, Mar 1999. Order no. 245125-001.
- An Introduction to the Intel® QuickPath Interconnect*. Intel Corporation, Mar 2010a. Reference no. 323535-001.
- Minimal Intel® Architecture Boot Loader—Bare Bones Functionality Required for Booting an Intel® Architecture Platform*. Intel Corporation, Jan 2010b. Reference no. 323246.
- Intel® Core 2 Duo and Intel® Core 2 Solo Processor for Intel® Centrino® Duo Processor Technology Intel® Celeron® Processor 500 Series - Specification Update*. Intel Corporation, Dec 2010c. Reference no. 314079-026.
- Intel® architecture Platform Basics*. Intel Corporation, Sep 2010d. Reference no. 324377.
- Intel® Trusted Execution Technology (Intel® TXT) LAB Handout*. Intel Corporation, 2010e. [Online; accessed 2-July-2015].
- Intel® Xeon® Processor 7500 Series Uncore Programming Guide*. Intel Corporation, Mar 2010f. Reference no. 323535-001.
- Intel® 7 Series Family - Intel® Management Engine Firmware 8.1 - 1.5MB Firmware Bring Up Guide*. Intel Corporation, Jul 2012a. Revision 8.1.0.1248 - PV Release.
- Intel® Xeon® Processor E5-2600 Product Family Uncore Performance Monitoring Guide*. Intel Corporation, Mar 2012b. Reference no. 327043-001.
- Software Guard Extensions Programming Reference*. Intel Corporation, 2013. Reference no. 329298-001US.
- Intel® Xeon® Processor 7500 Series Datasheet - Volume Two*. Intel Corporation, Mar 2014a. Reference no. 329595-002.
- Intel® Xeon® Processor E7 v2 2800/4800/8800 Product Family Datasheet - Volume Two*. Intel Corporation, Mar 2014b. Reference no. 329595-002.
- Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, Sep 2014c. Reference no. 248966-030.
- Software Guard Extensions Programming Reference*. Intel Corporation, 2014d. Reference no. 329298-002US.

- Intel® 100 Series Chipset Family Platform Controller Hub (PCH) Datasheet - Volume One*. Intel Corporation, Aug 2015a. Reference no. 332690-001EN.
- Mobile 4th Generation Intel® Core® Processor Family I/O Datasheet*. Intel Corporation, Feb 2015b. Reference no. 329003-003.
- Intel® Xeon® Processor E5-1600, E5-2400, and E5-2600 v3 Product Family Datasheet - Volume Two*. Intel Corporation, Jan 2015c. Reference no. 330784-002.
- Intel® Xeon® Processor 5500 Series - Specification Update*. Intel Corporation, 2 2015d. Reference no. 321324-018US.
- Intel® Xeon® Processor E5 Product Family - Specification Update*. Intel Corporation, Jan 2015e. Reference no. 326150-018.
- Intel® Software Guard Extensions (Intel® SGX)*. Intel Corporation, Jun 2015f. Reference no. 332680-002.
- Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, Sep 2015g. Reference no. 325462-056US.
- Intel® C610 Series Chipset and Intel® X99 Chipset Platform Controller Hub (PCH) Datasheet*. Intel Corporation, Oct 2015h. Reference no. 330788-003.
- Bruce Jacob and Trevor Mudge. Virtual memory: Issues of implementation. *Computer*, 31(6):33–43, 1998.
- 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.
- Jakob Jonsson and Burt Kaliski. RFC 3447: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. *Internet RFCs*, Feb 2003.
- Burt Kaliski. RFC 2313: PKCS #1: RSA Encryption Version 1.5. *Internet RFCs*, Mar 1998.
- Burt Kaliski and Jessica Staddon. RFC 2437: PKCS #1: RSA Encryption Version 2.0. *Internet RFCs*, Oct 1998.
- Corey Kallenberg, Xeno Kovah, John Butterworth, and Sam Cornwell. Extreme privilege escalation on windows 8/UEFI systems, 2014.

- Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 1–17. Springer, 2009.
- Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC Press, 2014.
- 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.
- Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX Annual Technical Conference*, pages 139–144, 2013.
- 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 Architecture*, pages 361–372. IEEE Press, 2014.
- L. A. Knauth and P. J. Ireland. Apparatus and method for providing eventing ip and source data address in a statistical sampling infrastructure, 2014. US Patent App. 13/976,613.
- N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.
- Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO)*, pages 388–397. Springer, 1999.
- 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.
- Hugo Krawczyk, Ran Canetti, and Mihir Bellare. HMAC: Keyed-hashing for message authentication. 1997.
- Markus G. Kuhn. Electromagnetic eavesdropping risks of flat-panel displays. In *Privacy Enhancing Technologies*, pages 88–107. Springer, 2005.
- Tsvika Kurts, Guillermo Savransky, Jason Ratner, Eilon Hazan, Daniel Skaba, Sharon Elmosnino, and Geeyarapuram N. Santhanakrishnan. Generic debug eXternal connection (gdx) for high integration integrated circuits, 2011. US Patent 8,074,131.
- David Levinthal. Performance analysis guide for Intel® Core i7 processor and Intel® Xeon 5500 processors. [https://software.intel.com/sites/products/collateral/hpc/vtune/performance\\_analysis\\_guide.pdf](https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf), 2010. [Online; accessed 26-January-2015].

- David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.
- 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.
- Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *ACM Sigplan Notices*, volume 9, pages 50–59. ACM, 1974.
- 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.
- Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiawicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324. ACM, 2013.
- James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS# 1 v2.0. In *Advances in Cryptology – CRYPTO 2001*, pages 230–238. Springer, 2001.
- 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.
- Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.
- David McGrew and John Viega. The galois/counter mode of operation (GCM). 2004. [Online; accessed 28-December-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, Stalinselvvaraj 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.
- Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.
- National Institute of Standards and Technology (NIST). The advanced encryption standard (AES). *Federal Information Processing Standards (FIPS) Publications (PUBS)*, 197, Nov 2001.
- National Institute of Standards and Technology (NIST). The digital signature standard (DSS). *Federal Information Processing Standards (FIPS) Processing Standards Publications (PUBS)*, 186-4, Jul 2013.
- National Security Agency (NSA) Central Security Service (CSS). Cryptography today on suite B phase-out. [https://www.nsa.gov/ia/programs/suiteb\\_cryptography/](https://www.nsa.gov/ia/programs/suiteb_cryptography/), Aug 2015. [Online; accessed 28-December-2015].
- M. S. Natsu, S. Datta, J. Wiedemeier, J. R. Vash, S. Kottapalli, S. P. Bobholz, and A. Baum. Supporting advanced RAS features in a secured computing system, 2012. US Patent 8,301,907.
- 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.
- Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer, 2006.
- Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO (extended version). *University of Cambridge, Computer Laboratory, Technical Report*, (UCAM-CL-TR-745), 2009.
- Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. ACCessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, page 9. ACM, 2012.

- D. B. Papworth, G. J. Hinton, M. A. Fetterman, R. P. Colwell, and A. F. Glew. Exception handling in a processor that performs speculative out-of-order instruction execution, 1999. US Patent 5,987,600.
- David A. Patterson and John L. Hennessy. *Computer Organization and Design: the hardware/software interface*. Morgan Kaufmann, 2013. ISBN 978-0-12-374750-1.
- P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. Reverse engineering Intel DRAM addressing and exploitation. *ArXiv e-prints*, Nov 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.
- S. A. Qureshi and M. O. Nicholes. System and method for using a firmware interface table to dynamically load an ACPI SSDT, 2006. US Patent 6,990,576.
- S. Raikin and R. Valentine. Gather cache architecture, 2014. US Patent 8,688,962.
- S. Raikin, O. Hamama, R. S. Chappell, C. B. Rust, H. S. Luu, L. A. Ong, and G. Hildesheim. Apparatus and method for a multiple page size translation lookaside buffer (TLB), 2014. US Patent App. 13/730,411.
- Stefan Reinauer. x86 Intel: Add firmware interface table support. <http://review.coreboot.org/#/c/2642/>, 2013. [Online; accessed 2-July-2015].
- 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.
- R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
- S. D. Rodgers, K. K. Tiruvallur, M. W. Rhodehamel, K. G. Konigsfeld, A. F. Glew, H. Akkary, M. A. Karnik, and J. A. Brayton. Method and apparatus for performing operations based upon the addresses of microinstructions, 1997. US Patent 5,636,374.
- S. D. Rodgers, R. Vidwans, J. Huang, M. A. Fetterman, and K. Huck. Method and apparatus for generating event handler vectors based on both operating mode and event type, 1999. US Patent 5,889,982.



- M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, May 2005.
- Xiaoyu Ruan. *Platform Embedded Security Technology Revealed*. Apress, 2014. ISBN 978-1-4302-6571-9.
- Joanna Rutkowska. Intel x86 considered harmful. [https://blog.invisiblethings.org/papers/2015/x86\\_harmful.pdf](https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf), Oct 2015. [Online; accessed 2-Nov-2015].
- Joanna Rutkowska and Rafał Wojtczuk. Preventing and detecting Xen hypervisor subversions. *Blackhat Briefings USA*, 2008.
- Jerome H. Saltzer and M. Frans Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann, 2009.
- 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 and S. J. Robinson. Enabling virtualization of a processor resource, 2014. US Patent 8,806,104.
- Stephen Shankland. Itanium: A cautionary tale. Dec 2005. [Online; accessed 11-February-2015].
- Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3): 473–530, 1982.
- Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.
- Sean W. Smith, Ron Perez, Steve Weingart, and Vernon Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*. IBM Thomas J. Watson Research Division, 1999.
- Marc Stevens, Pierre Karpman, and Thomas Peyrin. Free-start collision on full SHA-1. Cryptology ePrint Archive, Report 2015/967, 2015.
- 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<sup>nd</sup> 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.
- Trusted Computing Group TCG. Tpm main specification. [http://www.trustedcomputinggroup.org/resources/tpm\\_main\\_specification](http://www.trustedcomputinggroup.org/resources/tpm_main_specification), 2003.
- Alexander Tereshkin and Rafal Wojtczuk. Introducing ring-3 rootkits. Master’s thesis, 2009.
- Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proceedings of the 28th European Solid-State Circuits Conference (ESSCIRC)*, pages 403–406. IEEE, 2002.
- Unified Extensible Firmware Interface Specification, Version 2.5*. UEFI Forum, 2015. [Online; accessed 1-Jul-2015].
- Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.
- Wim Van Eck. Electromagnetic radiation from video display units: an eavesdropping risk? *Computers & Security*, 4(4):269–286, 1985.
- Amit Vasudevan, Jonathan M. McCune, Ning Qu, Leendert Van Doorn, and Adrian Perrig. Requirements for an integrity-protected hypervisor on the x86 hardware virtualized architecture. In *Trust and Trustworthy Computing*, pages 141–165. Springer, 2010.
- Sathish Venkataramani. *Advanced Board Bring Up - Power Sequencing Guide for Embedded Intel Architecture*. Intel Corporation, Apr 2011. Reference no. 325268.
- Vassilios Ververis. Security evaluation of Intel’s active management technology. 2010.
- Filip Wecherowski. A real SMM rootkit: Reversing and hooking BIOS SMI handlers. *Phrack Magazine*, 13(66), 2009.
- 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 and Alexander Tereshkin. Attacking Intel® BIOS. *Invisible Things Lab*, 2010.
- Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another way to circumvent Intel® trusted execution technology. *Invisible Things Lab*, 2009.
- Y. Wu and M. Breternitz. Genetic algorithm for microcode compression, 2008. US Patent 7,451,121.
- Y. Wu, S. Kim, M. Breternitz, and H. Hum. Compressing and accessing a microcode ROM, 2012. US Patent 8,099,587.
- 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.
- A. C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 162–167, 1986.
- 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. *Using secure coprocessors*. PhD thesis, Carnegie Mellon University, 1994.
- Marcelo Yaffe, 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.
- Xiantao Zhang and Yaozu Dong. Optimizing Xen VMM based on Intel® virtualization technology. In *Internet Computing in Science and Engineering, 2008. ICICSE'08. International Conference on*, pages 367–374. IEEE, 2008.
- Li Zhuang, Feng Zhou, and J. Doug Tygar. Keyboard acoustic emanations revisited. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):3, 2009.

- V. J. Zimmer and S. H. Robinson. Methods and systems for microcode patching, 2012. US Patent 8,296,528.
- V. J. Zimmer and J. Yao. Method and apparatus for sequential hypervisor invocation, 2012. US Patent 8,321,931.